Marco Campion

Partial (In)Completeness in Abstract Interpretation

Ph.D. Thesis

Università degli Studi di Verona
Dipartimento di Informatica

Partial (In)Completeness in Abstract Interpretation

Marco Campion

Ph.D. Thesis

Advisor: Prof. Roberto Giacobazzi

Università degli Studi di Verona
Dipartimento di Informatica
Strada le Grazie 15, 37134 Verona
Italy

I, Marco Campion, declare that this thesis titled, "Partial (In)Completeness in Abstract Interpretation" and the work presented in it are my own. I confirm that: (i) This work was done wholly or mainly while in candidature for a research degree at this University. (ii) Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated. (iii) Where I have consulted the published work of others, this is always clearly attributed. (iv) Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work. (v) I have acknowledged all main sources of help. (vi) Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Verona, 29th September 2021

ACKNOWLEDGMENTS

First and foremost I am extremely grateful to my supervisor Prof. Roberto Giacobazzi and to Prof. Mila Dalla Preda for their invaluable advice, continuous support and patience during my PhD study. Their immense knowledge and plentiful experience have encouraged me in all the time of my academic research and daily life. This thesis is the result of three years of research activity and it would not have been possible without their support and expertise on the subject.

I would also like to thank my PhD thesis referees Xavier Rival and Pierre Ganty for their deep understanding of my thesis and their extremely positive reviews with useful comments that helped to improve this manuscript.

ABSTRACT

In the abstract interpretation framework, completeness represents an optimal simulation by the abstract operators over the behavior of the concrete operators. This corresponds to an ideal (often rare) feature where there is no loss of information accumulated in abstract computations with respect to the properties encoded by the underlying abstract domains. In this thesis, we deal with the opposite notion of completeness in abstract interpretation, that is, incompleteness, applied to two different contexts: static program analysis and formal languages over the Chomsky's hierarchy.

In static program analysis, completeness is a very rare condition to be satisfied in practice and only the straightforward abstractions are complete for all programs, thus, we usually deal with incompleteness. For this reason, we introduce the notion of partial completeness. Partial completeness is a weaker notion of completeness which requires the imprecision of the analysis to be limited. A partially complete abstract interpretation allows some false alarms to be reported, but their number is bounded by a constant. We collect in partial completeness classes all the programs whose abstract interpretations share the same upper bound of imprecision. We then focus on the investigation of the computational limits of the class of partially complete programs with respect to a given abstract domain. Moreover, we show that the class of all partially complete programs is non-recursively enumerable, and its complement is productive whenever we allow an unlimited imprecision in the abstract domain. Finally, we formalize the local partial completeness class within which we require partial completeness only on some specific inputs. We prove that this last class of programs is a recursively enumerable set under a structural hypothesis on the underlying abstract domain, by showing an algorithm capable of proving the local partial completeness of a program with respect to a given abstract domain and an upper bound of imprecision.

In formal language theory, we want to study a possible reformulation, by abstract interpretation, of classes of languages in the Chomsky's hierarchy, and,

by exploiting the incompleteness of languages abstractions, we want to define separation results between classes of languages. To this end, we do a first step into this direction by studying the relation between indexed languages (recognized by indexed grammars) and context-free languages. Indexed grammars are a generalization of context-free grammars which recognize a proper subset of context-sensitive languages, the so called indexed languages. For example, indexed grammars can recognize the language $\{a^n b^n c^n \mid n \geq 1\}$ which is not context-free, but they cannot recognize $\{(ab^n)^n \mid n \geq 1\}$ which is context-sensitive. Indexed grammars identify a set of languages that are more expressive than context-free languages, while having decidability results that lie in between the ones of context-free and context-sensitive languages. We provide a fixpoint characterization of the languages recognized by an indexed grammar and we study possible ways to abstract, in the abstract interpretation sense, these languages and their grammars into context-free and regular languages. We formalize the separation class between indexed and context-free languages, i.e., all the languages that cannot be generated by a context-free grammar, as an instance of incompleteness of stack elimination abstraction over indexed grammars.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ASSUMPTIONS

Chapter 1

INTRODUCTION

The theory of *Abstract Interpretation*, introduced by Cousot and Cousot [26,27], is a general theory for the approximation of dynamic systems. It generalizes most existing methodologies for analyzing the semantic behaviors of a system, for example, a program, into a unique sound-by-construction framework which is based on a simple bu striking idea: *extracting program properties of a system is approximating its semantics* [26]. That is, the starting semantics, also called *concrete semantics*, is approximated by another semantics, called *abstract semantics*, by substituting the concrete domain of computation and its concrete semantic operations with, respectively, an *abstract domain* and *abstract semantic operations*. The basic intuition is that abstract domains are representations of some properties of interest about concrete domains' values, while abstract operations simulate, over the properties encoded by the abstract domains, the behavior of their concrete counterparts. The notion of approximation is encoded by suitable partial orderings $\leq$ on domains' objects, where the concrete and abstract domains are assumed to be partial ordered sets. That is, if x and y are two elements of a generic domain C, then $x \leq_C y$ means that x is more precise that y, or y carries less information than x. Therefore, as a very basic requirement, concrete and abstract operations preserve the approximation orderings. Two essential properties about abstract interpretation are correctness and completeness. *Correctness* is a fundamental condition of any approximation technique, and this holds also for abstract interpretation: if a concrete value is approximated by an abstract value, then a concrete computation step of that concrete value is still approximated by an abstract computation step of its abstract value with a possible loss of information that may occurred in simulating the behavior of the concrete operator by the abstract operator. Conversely, *Completeness*, also known in the literature with the terms *exactness* or *faithfulness* or *optimality*, is a stronger property which means that, relatively to the semantics

properties encoded by the abstract domains, such losses of information *never* occur. While soundness is the basic requirement for any abstract interpretation, the completeness property of an abstract semantic operator is not guaranteed by the abstract interpretation framework because it is a domain property, namely, it can be reached by refining the abstract domain considered [43]. Indeed, completeness is an ideal (and uncommon) situation where, informally, the abstract semantics is able to take full advantage of the expressive power of the underlying abstract domains. That is why completeness plays a central role in the abstract interpretation framework and many works devoted their attention to it. It has been first considered by Cousot and Cousot [27] who studied the basic properties of complete abstract interpretations and gave some examples in data-flow analysis. Although in some topics concerning abstract interpretation, completeness is *typically recurrent*, e.g., in comparative semantics [24, 25] where the studying of formal semantics at different levels of abstraction the corresponding computational domains are sufficiently rich of information so that completeness holds, there are some areas where completeness is *highly desirable*, for example in abstract model-checking [16, 34, 57] and *static program analysis* [8, 44], while in other areas its contribution is still *unclear*, for example with the purpose of separating known classes of *formal languages* in Chomsky's hierarchy [11].

1.1 Motivation

Static Program Analysis

Static program analysis allows us to reason on the behavior of programs without actually running them. Indeed, the aim of static analysis is to provide answers to queries on the concrete program behavior by reasoning on an abstract program behavior. Abstraction is needed in order to design sound-by-construction static analyzers that provide an approximated answer to queries that deal with arbitrary extensional properties of programs (e.g. absence of runtime errors, variables values that do not overflow, etc.). The static analysis of a program P begins with the design of an abstract analyzer, i.e., an abstract domain A that expresses some behavioral property of interest, and an interpreter for our language, defined on that abstract domain [26, 38]. Programs are associated with abstract denotations on the abstract domain A, which is a partially ordered set where the partial order $\leq_A$ encodes the relative precision of its elements. Given a program P and an abstract domain A, a static analyzer can verify any query q that can be precisely expressed on the abstract domain A, namely an element of A. Static analyzers establish whether the behavior of program P interpreted on the abstract domain A, denoted $[\![P]\!]^A$, satisfies a certain query $q \in A$, namely

whether $[\![P]\!]^A \leq_A q$. By abstract interpretation, the static analysis is always sound: if program P satisfies the abstract check $[\![P]\!]^A \leq_A q$ then the concrete behavior of P, denoted by $[\![P]\!]$, also satisfies the query q. However, the converse does not hold in general and it may happen that the abstract interpretation of the behavior of P does not satisfy the query q, i.e., $[\![P]\!]^A \not\leq_A q$ while the concrete program behavior $[\![P]\!]$ satisfies q. This happens when the imprecision implicit in the abstract program behavior $[\![P]\!]^A$ gives rise to false alarms.

A major part (and cost) of this design is the tuning of the *efficiency/precision trade-off*—the analysis should deliver the expected answer on a set of programs of interest T without costing too much computational time and/or space resources. To achieve this, the designer gives up some of the precision for programs outside T. The rationale for this design choice is that T represents programs of interest for the designer/user and for which the analysis has to be precise. Widening T may represent a problem as uncontrolled imprecision may arise. The tension between precision and scalability is clear here: As all alarming systems, a program analysis tool is credible—therefore usable—when few false-alarms are reported.

Formal Languages

Abstract interpretation was originally developed as a unifying framework for designing and validating static (i.e., at compile time) program analysis, but it gained popularity as a general methodology for describing and formalizing approximate computations in many different areas of computer science, for example, in model checking [16, 34, 57], type inference [19, 60], malware detection [33], dynamic analysis [66], grammar structures [5, 25, 28], formal languages [11, 17, 36, 40].

We want to exploit this latter line of research by establishing a relation, formalized by abstract interpretation, between indexed grammars, which are a formalism describing indexed languages, with the aim of relating languages in *Chomsky's hierarchy* [15]. Chomsky's hierarchy drove most of the research in theoretical computer science for decades. Its structure, and its inner separation results between formal languages, represent the corner stone to understand the expressive power of symbolic structures. The three well-known recursive families of formal languages in the Chomsky's hierarchy are (ordered by strict inclusion): regular languages, generated by type-3 grammars (left and right-linear grammars) and accepted by finite state automata, context-free languages, generated by type-2 grammars (also called context-free grammars) and accepted by non-deterministic pushdown automata, and context-sensitive languages, generated by type-1 grammars (also called context-sensitive grammars) and accepted by linear-bounded non-deterministic Turing machines. *Indexed languages* have

been introduced in [2] as an extension of context-free languages in order to include languages such as $\{a^n b^n c^n \mid n \geq 1\}$. It is known that indexed languages are strictly less expressive than context-sensitive languages, e.g., the language $\{(ab^n)^n \mid n \geq 1\}$ is context-sensitive but not indexed. This intermediate class between context-free and context-sensitive has interesting properties, e.g., decidable emptiness test and NP-complete membership check, where the first is undecidable and the latter is PSPACE-complete in the family of context-sensitive languages. Indexed languages are recognized by nested stack automata [3] and they are described by *indexed grammars* which differ from context-free grammars in that each non-terminal is equipped with a stack on which push and pop instructions can be performed. Moreover, the stack can be copied to all non-terminals on the right side of each production.

For our purposes, indexed languages represent an ideal concrete semantics for rebuilding part of Chomsky's hierarchy by abstract interpretation, in particular for the case of regular and context-free languages. This is due to two reasons: (1) they lack, to the best of our knowledge, of a fixpoint semantics and (2) they represent an intermediate family of languages between context-free and context-sensitive, therefore including context-free and regular languages as subclasses.

1.2 The Problem

Static Program Analysis

Completeness in static program analysis by abstract interpretation means *precision*. It represents the ideal situation where no false alarms are produced when answering queries on program behavior by means of an abstract interpreter. Completeness, however, is a *very rare* condition to be satisfied in practice. Although we can refine our abstract domain in order to reach completeness for the analysis of a given program [43], abstraction refinement techniques do not solve the problem since they often lead to very concrete abstract domains that are too costly to use, therefore breaking efficiency. However, experience tells us that there are results that are "more incomplete" than others. Consider for example the two programs P and Q defined, respectively, in Figures 1.1, 1.2. These programs are equivalent: they both manipulate the value of variable x that at the end of the computation holds the value 0. Let us consider a query q that checks whether the value of variable x at the end of computation is 0, denoted as $q : x \stackrel{?}{=} 0$. It is clear that the concrete program behavior of P and Q, denoted as $[\![P]\!]$ and $[\![Q]\!]$, satisfy the query q. However, it is easy to observe that we have incompleteness when we verify the query q for programs P and Q on the abstract domain $\mathsf{P} \sqcap \mathsf{S}$ shown in Figure 1.3, which is obtained as the reduced

$$
\begin{array}{l}
x := 0; \\
x := x + 1; \\
x := x - 1 \\
// \ q : x \stackrel{?}{=} 0
\end{array}
$$

Figure 1.1: The program P

$$
\begin{array}{l}
x := 2; \\
x := x - 1; \\
\textbf{if } x \geq 1 \textbf{ then } x := x - 1 \\
\textbf{else skip} \\
// \ q : x \stackrel{?}{=} 0
\end{array}
$$

Figure 1.2: The program Q

product between Sign [21] and Parity [21] abstract domains, both shown in Figure 3.1. Let $\alpha_{\mathsf{P}\sqcap\mathsf{S}} : \wp(\mathbb{Z}) \to \mathsf{P} \sqcap \mathsf{S}$ be the abstraction function that maps the sets of integer values in the element in $\mathsf{P} \sqcap \mathsf{S}$ that better approximates it. So, for example, $\alpha_{\mathsf{P}\sqcap\mathsf{S}}(\{-3,-7\}) = (odd, -)$, $\alpha_{\mathsf{P}\sqcap\mathsf{S}}(\{-3,7\}) = odd$ and so on. The query is correctly represented by the abstract domain $\mathsf{P}\sqcap\mathsf{S}$ since 0 is an element of this domain that precisely expresses the value 0 assumed by variable x. Indeed, if we interpret the result of the concrete computation on the abstract domain $\mathsf{P}\sqcap\mathsf{S}$, we obtain $\alpha_{\mathsf{P}\sqcap\mathsf{S}}(\llbracket P \rrbracket) = \alpha_{\mathsf{P}\sqcap\mathsf{S}}(\llbracket Q \rrbracket) = 0$, while if we interpret their behavior on the abstract domain $\mathsf{P}\sqcap\mathsf{S}$ we obtain $\llbracket P \rrbracket^{\mathsf{P}\sqcap\mathsf{S}} = even$, and $\llbracket Q \rrbracket^{\mathsf{P}\sqcap\mathsf{S}} = \mathbb{Z}$. Thus, in both cases we have incompleteness since $\alpha_{\mathsf{P}\sqcap\mathsf{S}}(\llbracket P \rrbracket) \neq \llbracket P \rrbracket^{\mathsf{P}\sqcap\mathsf{S}}$ and $\alpha_{\mathsf{P}\sqcap\mathsf{S}}(\llbracket P \rrbracket) \leq_{\mathsf{P}\sqcap\mathsf{S}} 0$ while $\llbracket P \rrbracket^{\mathsf{P}\sqcap\mathsf{S}} \not\leq_{\mathsf{P}\sqcap\mathsf{S}} 0$, and the same for program Q where $\alpha_{\mathsf{P}\sqcap\mathsf{S}}(\llbracket Q \rrbracket) \neq \llbracket Q \rrbracket^{\mathsf{P}\sqcap\mathsf{S}}$ and $\alpha_{\mathsf{P}\sqcap\mathsf{S}}(\llbracket Q \rrbracket) \leq_{\mathsf{P}\sqcap\mathsf{S}} 0$ while $\llbracket Q \rrbracket^{\mathsf{P}\sqcap\mathsf{S}} \not\leq_{\mathsf{P}\sqcap\mathsf{S}} 0$. In the considered example, the computation of program Q on $\mathsf{P}\sqcap\mathsf{S}$ is "*more incomplete*" than the computation of P on $\mathsf{P}\sqcap\mathsf{S}$, since the result that we obtain is "*more far from precision*": $\llbracket P \rrbracket^{\mathsf{P}\sqcap\mathsf{S}} = even$ is closer to 0 on the domain $\mathsf{P}\sqcap\mathsf{S}$ than $\llbracket Q \rrbracket^{\mathsf{P}\sqcap\mathsf{S}} = \mathbb{Z}$.

This difference among incompleteness results is what we would like to express and measure. The standard abstract interpretation framework does not allow us to reason and compare the degree of imprecision of the results of incomplete analyses. There are some related preliminary results in this direction as for example the definition of a domain-specific measure of imprecision of static analysis [13, 56, 71] or the new formalization of the classical abstract interpre-

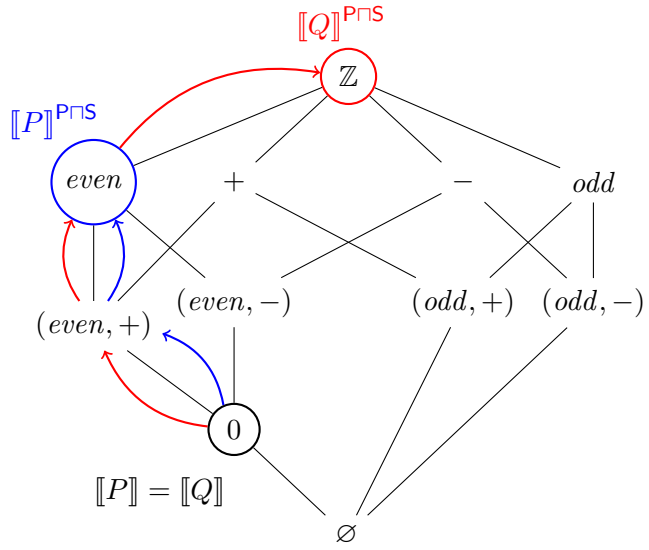


Figure 1.3: P ⊓ S abstract domain

tation framework [31, 37] in order to somehow embody the concept of closeness between approximations. Moreover, there are recent works that study the recursive properties of the classes of programs that are complete for a given abstract interpreter [44] in order to better understand the boundaries of (im)precision of the abstract interpreter. However, there is no general definition for the concept of bounded imprecision in abstract interpretation, i.e., where we can limit the amount of false alarms in the result of the analysis.

Formal Languages

Although in the literature there are some works that approximate grammar structures by abstract interpretation [5] and apply specific language abstractions for specific purposes, like the design of static analyzers [25, 28], formal verification [36], and for the language inclusion problem [40], no approaches considered the more general problem of correlating languages in Chomsky's hierarchy by the theory of fixpoint abstraction by abstract interpretation. Furthermore, the classes of regular, context-free and context-sensitive languages are not closed under intersection. This implies that it is not possible to specify Galois insertions between the domains of languages in the Chomsky's hierarchy. However, this does not exclude approximation strategies on the fixpoint semantics on the underlying grammar generating the considered languages, by acting on the productions, which can be considered as abstracting the concrete semantics (i.e., the

productions) of the starting grammar in order to get an approximated semantics (namely, a grammar with new productions) that generates a less expressive language including the former respect to the standard set inclusion. We will see that, in our settings, we need *incomplete abstractions* in order to get a strict approximation of grammars that generates a less expressive formal languages, because otherwise, a complete abstraction means no modification on the underlying grammar structure. This means that incompleteness plays a special rule in our grammar abstractions and can be exploited in order to form separation results between family of formal languages.

1.3 Contribution

Static Program Analysis

Our aim is to *weaken* the notion of completeness in static program analysis by abstract interpretation, into a less strong requirement, i.e., by letting imprecision but by a limited amount. In order to face this problem, we need to introduce a notion of distance A-compatible on the elements of an abstract domain A. We use a weaker form of metric function that is specific for the elements of an abstract domain, namely it is compatible with the underlying ordering relation and it takes into account the presence of uncomparable elements. Referring to the previous example, we can consider the distance in $\mathsf{P} \sqcap \mathsf{S}$ to be the length of the minimum path between two comparable abstract elements, for example, the distance between *even* and 0 is 2. This means that, the distance between the abstract computation of $[\![P]\!]^{\mathsf{P}\sqcap\mathsf{S}} = \mathit{even}$ and the concrete one $[\![P]\!] = 0$ is 2 (the blue arrows in Figure 1.3), while the distance between $[\![Q]\!]^{\mathsf{P}\sqcap\mathsf{S}} = \mathbb{Z}$ and $[\![Q]\!] = 0$ is 3 (the red arrows in Figure 1.3). This formalizes the intuition that the abstract computation of Q is "way more far from precision" then the one of P. This allows us to formalize the notion of *ε-partial completeness* of an abstract domain A endowed with a A-compatible distance, with respect to a given program, where the imprecision in the abstract analysis is bounded by ε, namely, the distance between the results of the concrete and abstract analysis on the considered program is at most ε. Given an abstract domain A, we collect the set of all programs whose analysis on A is ε-partial complete, and we call this set the ε-partial completeness class of the abstract domain A. This is a generalization of the completeness class of programs whose analysis on A is complete introduced in [44]. Indeed, the completeness classes correspond to the special case of 0-partial completeness. We investigate the properties of the ε-partial completeness classes. Firstly, we highlight that partial completeness shares some simple properties with the completeness class: they are both infi-

nite sets and in general non-extensional sets, i.e., not an index set of partial recursive functions. For the latter property, it is enough to consider the example above as a witness. Then, we show that the distance we choose for measuring the relative imprecision between abstract elements, plays an important role in the considered class of partial completeness. Indeed, we prove that, if for all $n \in \mathbb{Q}_{\geq 0}$ we can always find an element whose distance from $\bot_A$ is strictly greater than n, then we can always build a program that is not in the n-partial completeness class. An abstract domain satisfying the above condition will be said holding *unlimited imprecision*. Then, we prove that under non restrictive hypothesis for abstract interpretations, such as employing non trivial abstract domains (i.e., the abstract domain is different from the concrete domain) with unlimited imprecision, then for all bound ε we fix to the allowed imprecision, the ε-partial completeness class and its complement are both non recursively enumerable sets. This means that we cannot automate the procedure of deciding whether an input program satisfies a given precision bound. Moreover, we show that the ε-partial incompleteness class where the abstract domains have unlimited imprecision, is a special non-recursively enumerable set and shares a structure that is similar to the set of Gödel numbers of true sentences in first-order arithmetic: it is a productive set. Finally, we weaken also the ε-partial completeness requirement with the aim to obtain a semi-decidable procedure to prove that property: we introduce the notion of local ε-partial completeness admitting ε-partial completeness only to a fixed set of inputs. This corresponds to a further weakening to the notion of completeness and, specifically, to local completeness which has been recently introduced by Bruni et al. in [9]. We prove that, under structural hypothesis on the considered abstract domain, there exists a semi-decidable procedure that proves if a program is locally ε-partial complete.

Formal Languages

We have seen that completeness in static analysis by abstract interpretation means no loss of information on the underlying abstract domain chosen. Conversely, incompleteness represents an, although correct, imperfect analysis in terms of precision due to either how the program code is written (the intensional properties of the program) or the chosen store abstraction or the fixed abstract semantics used. We study the counterpart consequences of incompleteness in language abstractions. We show how incompleteness of language abstractions can be interpreted in terms of classes of formal languages in the Chomsky's hierarchy. Our intention is to generalize known *separation results* between classes of languages, e.g., the Pumping Lemmata, as *instances of incompleteness* of language abstractions. The idea is that, if a family of languages corresponds to a suitable abstraction of the fixpoint semantics of a more concrete family of

languages, then languages not expressible in one family should correspond to *witnesses of the incompleteness* of this abstraction. The interest in this perspective over Chomsky's hierarchy is in the fact that we would be able to reformulate most of this hierarchy, including separation results, in terms of abstract interpretation, providing powerful tools for comparing symbolic abstract domains with respect to their expressive power. We provide a first step into this direction, by constructing the separation class of languages between indexed and context-free languages using incompleteness of stack elimination abstraction, that is, the abstraction that over approximates each stack of a non-terminal symbol to all possible stack configurations in an indexed grammar.

In order to reach our goal, we first construct a fixpoint semantics for indexed languages. The construction follows the one known for CS languages, and derives a system of equations associated with each indexed grammar. We prove that the fixpoint solution of this system of equations corresponds precisely to the language generated by the grammar. This will provide the base fixpoint semantics for making abstract interpretation. We show that no best abstraction, which in abstract interpretation are represented by Galois Insertions, is possible between indexed languages and respectively context-free and regular languages, w.r.t. set inclusion. This means that we need to act at the level of grammar structures (i.e., on the way languages are generated and represented in grammatical form) in order to generate languages as abstract interpretations of an indexed grammar. We introduce several abstractions of grammatical structures in such a way that the abstract language transformer associated with the system of equations of the indexed language generates the desired language. We show that certain simplifications of the productions of indexed grammars can be specified as abstractions, now in the standard Galois insertion based framework, and that the corresponding abstract semantics coincides precisely to classes of languages in Chomsky's hierarchy, in our case the class of context-free languages. The main advantage is that known fixpoint characterization and algorithms for context-free languages can be extracted in a *calculational* way by abstract interpretation of the fixpoint semantics of the more concrete indexed grammars. This shows that standard methods for the design of static program analyses and hierarchy of semantics (e.g., see [20,28,29]) can be applied to systematically derive fixpoint presentations for families of formal languages and to let abstract interpretation methods to be applicable to Chomsky's hierarchy. Finally, we show that we can build the separation class between context-free and indexed languages by exploiting the incompleteness of stack elimination abstraction, thus providing a first step towards reformulating Chomsky's hierarchy by abstract interpretation.

1.4 Structure of the Thesis

This thesis is structured as follows. In the first two chapters, we aim to provide the reader with all the necessary background for reading this thesis. Moreover, we will highlight the major assumptions that will affect the main results of this thesis. Chapter 2 provides notation and the basic algebraic notions that we are going to use in the following of the thesis. We define a collecting denotational program semantics over a standard deterministic while-language Prog with integer and Boolean expressions and no runtime errors. We then recall computability notions applied to (sets of) programs and stores defined in the previous section. Finally, we recall the definitions of the three most common formal languages in the Chomsky's hierarchy: regular, context-free and context-sensitive language families.

In Chapter 3 we give a brief introduction to the abstract interpretation framework. In particular, in the context of static program analysis, we define store abstractions and the abstract semantics of while programs in Prog. We rigorously define trivial and recursive abstract domains of stores and the completeness class of programs induced by an abstract domain of stores.

In Chapter 4 we present the first main contribution of the thesis. We define our distance model for partial completeness classes. We outline the definition of quasi-metric A-compatible, that is, a quasi-metric that satisfies specific axioms in order to be connected to the structure of any recursive abstract domains. We introduce the concept of ε-partial completeness of a program with respect to a given abstract domain A and a quasi-metric A-compatible. We then study the computational limits of this new framework by proving that the set of all ε-partial complete programs is non-recursively enumerable and the complement set is productive under some assumptions on the underlying abstract domain of stores. Finally, with the aim of finding a decidable partial completeness class, we define the local ε-partial completeness and we show that it is recursively enumerable if the abstract domain of stores A satisfies a specific structural property.

Chapter 5 shows how abstract interpretation framework can be applied to formal languages in order to characterize the separation classes between families of languages. Indexed grammars and languages are defined as an intermediate formal language between context-free and contest-sensitive languages. We provide a fixpoint characterization of indexed languages by deriving a system of equations from the underlying indexed grammars. We then provide some indexed grammar abstractions by acting on the grammar productions in order to obtain a less expressive formal language. Finally we show how we can use incompleteness of indexed grammar abstractions in order to build the separation class between indexed and context-free languages.

Chapter 6 compares our results to prior results, while Chapter 7 sums up the major contributions of this thesis and briefly describes future works that we would like to explore.

Chapter 2

BACKGROUND

Before starting with the main results of this thesis, in this and the next chapter we recall some basic concepts and fix notation. We start with functions and lattices definitions in Section 2.1, then we move to the collecting denotational semantics of a simple imperative while-language in Section 2.2. In Section 2.3, we recall some computability notions using the enumerable set of stores $\mathbb{S}$ as an equivalent substitute to the set of natural numbers $\mathbb{N}$ for the domain of partial recursive functions. Finally, in Section 2.4 we review some formal language theory.

2.1 Order Theory

Sets and relations. Given two sets S and T, $\wp(S)$ denotes the powerset of S, $S \smallsetminus T$ denotes the set-difference between S and T, $\overline{S}$ denotes the complement of S with respect to some universe set determined by the context, $S \subseteq T$ denotes sets inclusion while $S \subset T$ (or $S \subsetneq T$) denotes strict sets inclusion, $|S|$ denotes the cardinality where S is finite if $|S| < \omega$, countably infinite if $|S| = \omega$, countable if $|S| \leq \omega$. A binary relation $\sim$ over sets S and T is a subset of the Cartesian product $\sim \subseteq S \times T$, that is, it is a set of ordered pairs (x, y) consisting of elements $x \in S$ and $y \in T$. A binary relation $\sim$ on a set S is said to be an equivalent relation if and only if, for all $x, y, z \in S$, it is reflexive ($x \sim x$), symmetric ($x \sim y \Leftrightarrow y \sim x$) and transitive ($x \sim y$ and $y \sim z$ implies $x \sim z$). Given a set S and an equivalence relation $\sim \subseteq S \times S$ on it, the equivalence class of an element $a \in S$, denoted $[a]_\sim$, is the set $[a]_\sim \triangleq \{x \in S \mid x \sim a\}$ of elements which are equivalent to a. The equivalence classes form a partition of S called the quotient set of S by $\sim$ and is denoted by $S_{/\sim}$. From now on, we will emphasize the set

S on which a binary relation $\sim$ is defined by the subscript $\sim_S$ except for the straightforward equivalence relation $=$ unless it has a different definition.

Functions. A function $f : S \to T$, where S is the domain and T the codomain of f, is injective if for all $x, y \in S$, $f(x) = f(y)$ implies $x = y$, is surjective if for every $y \in T$ there exists $x \in S$ such that $f(x) = y$, is a bijection if f is both injective and surjective. We denote with $f : S \to T$ a totally defined function and with $f : S \nrightarrow T$ a partially defined function. If $f : S \nrightarrow T$ then $f(x)\downarrow$ denotes that $f(x)$ is defined, $dom(f) \triangleq \{x \in S \mid f(x)\downarrow\}$ denotes the domain of f, given a subset $X \subseteq S$, $f(X) \triangleq \{f(x) \in T \mid x \in X \cap dom(f)\}$ denotes the image of f on X, where f is defined, while if $X = S$ then we call $range(f) \triangleq f(S)$ the range of f. Two partial functions $f, g : S \nrightarrow T$ are extensionally equivalent, denoted by $f \cong g$, if $dom(f) = dom(g)$ and for all $x \in dom(f) = dom(g)$, $f(x) = g(x)$. Sometimes we use a λ-notation $\lambda x.f(x)$ to emphasize the arguments of a function f. Given $f : S \nrightarrow T$ and $g : T \nrightarrow U$, $g \circ f : S \nrightarrow U$ denotes their composition, where $g \circ f(x) = g(f(x))$ when $f(x)\downarrow$ and $g(f(x))\downarrow$, otherwise $g \circ f$ is not defined on x.

Metric spaces. We denote with $\mathbb{N}$, $\mathbb{Q}$ and $\mathbb{R}$ the sets of all, respectively, natural, rational and real numbers. We will use subscripts in order to limit their scope, e.g., $\mathbb{Q}_{\geq 0}$ denotes the set of all non-negative rational numbers. A metric is a function that defines a distance between each pair of elements of a set S. Formally, a metric on a non-empty set S is a mapping $d : S \times S \to \mathbb{R}_{\geq 0}$ such that for every $x, y, z \in S$:

(i) $d(x, y) = 0 \Leftrightarrow x = y$ (identity of indiscernibles)

(ii) $d(x, y) = d(y, x)$ (symmetry)

(iii) $d(x, y) + d(y, z) \geq d(x, z)$ (triangle inequality)

where here and in the following, relations, functions and operations with no subscript are interpreted in the standard way depending on the context of application (in this case, real numbers). A set provided with a metric is called metric space. For example, the real numbers $\mathbb{R}$ with the distance function d between some $x, y \in \mathbb{R}$ such that $d(x, y) = |x - y|_{Abs}$ where $|\cdot|_{Abs}$ is the absolute difference, is a metric space.

Partial order sets. A partial order is a binary relation $\leq_L$ over a set L satisfying for every $x, y, z \in L$: reflexivity ($x \leq_L x$), antisymmetry ($x \leq_L y$ and $y \leq_L x$ implies $x = y$), transitivity ($x \leq_L y$ and $y \leq_L z$ implies $x \leq_L z$). A set L endowed with a partial order relation $\leq_L$ is called a partially ordered set, or briefly poset, and it is denoted by $\langle L, \leq_L \rangle$. We will use also the *strict* poset relation $<_L$ such that for any $x, y \in L$, $x <_L y$ iff $x \leq_L y$ and $x \neq y$. We say

that y *covers* x, written $x \lessdot_L y$, if $x < y$ and there is no element $z \in L$ such that $x < z < y$. Let $\langle L, \leq_L \rangle$ be a poset and $S \subseteq L$ be an arbitrary subset. $a \in S$ is a maximal element of S if for all $x \in S$: $a \leq_L x \Rightarrow a = x$, while a is maximum of S if for all $x \in S$, $x \leq_L a$. If L has maximum this is the top element and it will be denoted with $\top_L$. The bottom element is defined dually and denoted $\bot_L$. An element $u \in L$ is said to be an upper bound of S if $s \leq_L u$ for each $s \in S$. A set may have many upper bounds, or none at all. An upper bound u of S is said to be its least upper bound (lub), or join, or supremum, if $u \leq_L x$ for each upper bound x of S. The least upper bound of a set, when it exists, is unique. Dually, $l \in L$ is said to be a lower bound of S if $l \leq_L s$ for each $s \in S$. A lower bound l of S is said to be its greatest lower bound (glb), or meet, or infimum, if $x \leq_L l$ for each lower bound x of S. A set may have many lower bounds, or none at all, but can have at most one greatest lower bound.

Chains. A subset $Y \subseteq L$ of a poset $L = \langle L, \leq_L \rangle$ is a chain if for every $y_1, y_2 \in Y$, $y_1 \leq_L y_2$ or $y_2 \leq_L y_1$. Thus a chain is a (possibly empty) subset of L that is totally ordered. It is a finite chain if it is a finite subset of L. A sequence $(l_n)_{n \in \mathbb{N}} = \{l_n \mid n \in \mathbb{N}\}$ of elements in L is an ascending chain if $n \leq m$ implies $l_n \leq_L l_m$. Similarly, a sequence $(l_n)_n$ is a descending chain if $n \leq m$ implies $l_m \leq_L l_n$. Clearly, both ascending chains and descending chains are chains. A sequence $(l_n)_{n \in \mathbb{N}}$ eventually stabilizes if and only if there exists $n_0 \in \mathbb{N}$ such that for all $n \in \mathbb{N}$: $n \geq n_0$ implies $l_n = l_{n_0}$. A poset L has finite height if and only if all chains are finite. L satisfies the Ascending Chain Condition (ACC) if and only if all ascending chains eventually stabilize. Similarly, it satisfies the Descending Chain Condition (DCC) if and only if all descending chains eventually stabilize. L satisfies both the ACC and DCC if and only if it has finite height.

Lattices. A poset $\langle L, \leq_L \rangle$ is called a join-semilattice if each two-element subset $\{a, b\} \subseteq L$ has a join (i.e. least upper bound), and is called a meet-semilattice if each two-element subset has a meet (i.e. greatest lower bound), denoted by $a \vee_L b$ and $a \wedge_L b$ respectively. $\langle L, \leq_L \rangle$ is called a lattice if it is both a join- and a meet-semilattice. This definition makes $\vee_L$ and $\wedge_L$ binary operations. Both operations are monotone with respect to the given order: $a_1 \leq_L a_2$ and $b_1 \leq_L b_2$ implies that $a_1 \vee_L b_1 \leq_L a_2 \vee_L b_2$ and $a_1 \wedge_L b_1 \leq_L a_2 \wedge_L b_2$. A lattice $\langle L, \leq_L \rangle$ is complete when for all subsets $X \subseteq L$, arbitrary lubs $\bigvee_L X$ and glbs $\bigwedge_L X$ exist in L (empty subset included). A complete lattice L with partial order $\leq_L$, lub $\vee_L$, glb $\wedge_L$, greatest element (top) $\top_L$, and least element (bottom) $\bot_L$ is denoted by $\langle L, \leq_L, \vee_L, \wedge_L, \top_L, \bot_L \rangle$. A lattice $\langle L, \leq_L, \vee_L, \wedge_L \rangle$ is distributive if and only if for every $x, y, z \in L$ one of the following equivalent conditions is satisfied:

(i) $(x \wedge_L y) \vee_L (x \wedge_L z) = x \wedge_L (y \vee_L z)$;

(ii) $(x \vee_L y) \wedge_L (x \vee_L z) = x \vee_L (y \wedge_L z)$;

(iii) $(x \vee_L y) \wedge_L z \leq_L x \vee_L (y \wedge_L z)$.

Let $\langle L, \leq_L, \bot_L, \top_L \rangle$ be a poset with infimum $\bot_L$ and supremum $\top_L$. We say that $x \in L$ has a complement $y \in L$ if and only if $x \wedge_L y = \bot_L$ and $x \vee_L y = \top_L$. Note that in general the complement may not be unique. A complemented lattice is a lattice $\langle L, \leq_L, \vee_L, \wedge_L, \top_L, \bot_L \rangle$ in which every element $x \in L$ has a complement in L. A Boolean lattice is a complemented distributive lattice. A Boolean algebra $\langle L, \leq_L, \vee_L, \wedge_L, \neg_L, \top_L, \bot_L \rangle$ is a Boolean lattice in which $\leq_L, \top_L, \bot_L$ and $\neg_L$ are also considered as operations:

(i) $\langle L, \leq_L, \vee_L, \wedge_L \rangle$ is a distributive lattice;

(ii) for every $x, y \in L$: $x \leq_L y \triangleq x \vee_L y = y \Leftrightarrow x \wedge_L y = x$;

(iii) for all $x \in L$: $x \vee_L \bot_L = x$ and $x \wedge_L \top_L = x$;

(iv) $x \vee_L \neg_L x = \top_L$ and $x \wedge_L \neg_L x = \bot_L$.

For example, for each set S, let $\neg A \triangleq S \setminus A$ then $\langle \wp(S), \subseteq, \cup, \cap, \neg, S, \varnothing \rangle$ is a Boolean algebra called the powerset algebra.

Functions over posets. A function $f : L \to L$ over a poset $\langle L, \leq_L \rangle$ is monotone if for all $x, y \in L$ such that $x \leq_L y$, f preserves the order, i.e., $f(x) \leq_L f(y)$. Moreover, f is idempotent if for all $x \in L$, $f(f(x)) = f(x)$, increasing if $x \leq_L f(x)$. If $f, g : S \to L$ and $\langle L, \leq_L \rangle$ is a poset then the pointwise partial order relation is defined by: $f \sqsubseteq g$ when for all $x \in S$, $f(x) \leq_L g(x)$. If L is a (complete) lattice then $\langle S \to \langle L, \leq_L \rangle \rangle$ is a (complete) lattice. A function $f : L_1 \to L_2$ between complete lattices is additive (co-additive) if for all $Y \subseteq L_1$, $f(\vee_{L_1} Y) = \vee_{L_2} f(Y)$ ($f(\wedge_{L_1} Y) = \wedge_{L_2} f(Y)$). Also, f is continuous (co-continuous) when f preserves lubs (glbs) of chains in L_1.

Fixpoints. In mathematics, a fixpoint of a function is an element of the function's domain that is mapped to itself by the function, that is, $x \in L$ is a fixpoint of the function $f : L \to L$ if $f(x) = x$. The Knaster–Tarski theorem guarantees that if L is a complete lattice and $f : L \to L$ a monotone function, then the set of fixpoints of f in L is also a complete lattice. As a consequence, since complete lattices cannot be empty (they must contain supremum of empty set), the theorem in particular guarantees the existence of at least one fixpoint of f, and even the existence of a least (or greatest) fixpoint, denoted $\mathsf{lfp}(f)$ (resp. $\mathsf{gfp}(f)$). Moreover, if $f : L \to L$ is continuous then $\mathsf{lfp}(f) = \bigvee_{L\, n \in \mathbb{N}} f^n(\bot_L)$, where, for all $n \in \mathbb{N}$ and $x \in L$, f^n is inductively defined by: $f^0(x) \triangleq x$ and $f^{n+1}(x) \triangleq f(f^n(x))$.

$$\begin{aligned}
\mathrm{AExp} \ni a &::= v \in \mathbb{Z} \mid x_i \in \mathit{Var} \mid a + a \mid a - a \mid a * a \\
\mathrm{BExp} \ni b &::= \mathbf{t} \mid \mathbf{f} \mid a = a \mid a > a \mid b \wedge b \mid \neg b \\
\mathsf{Prog} \ni C &::= \mathbf{skip} \mid x := a \mid C; C \mid \mathbf{if}\ b\ \mathbf{then}\ C\ \mathbf{else}\ C \mid \mathbf{while}\ b\ \mathbf{do}\ C
\end{aligned}$$

Figure 2.1: Syntax of a while-language Prog

2.2 Semantics of While Programs

We consider a standard deterministic while-language Prog with integer and Boolean expressions and no runtime errors, as defined, e.g., in [75], whose syntax is defined in Figure 2.1. The finite set of variables occurring in some syntactic object c is denoted by $\mathit{vars}(c) \subseteq \mathit{Var}$. Var is assumed to be a denumerable set of variables and, without loss of generality, we define $\mathit{Var} \triangleq \{x_i \mid i \in \mathbb{N}_{>0}\}$, i.e., variables are indexed by positive integers. The index of variables is omitted in programs that have only one variable x. A program store for the variables in Var is a partial function $s \in \mathbb{S} \triangleq \mathit{Var} \nrightarrow \mathbb{Z}$, which can be equivalently specified by a tuple of its defined values $\langle x_1 \mapsto v_1, \ldots, x_n \mapsto v_n \rangle$. That is, the set of all stores is $\mathbb{S} \triangleq \bigcup_{n \in \mathbb{N}} \mathbb{Z}^n$. Store update is written $s[x_i \mapsto v]$. As usual, for some store function $s \in \mathbb{S}$, we let $\mathit{vars}(s) \triangleq \{x_i \in \mathit{Var} \mid s(x_i)\downarrow\}$, while for $S \subseteq \mathbb{S}$, we denote by $\mathit{vars}(S) \triangleq \bigcup_{s \in S} \mathit{vars}(s)$.

The semantics of arithmetic expressions is defined by the function $(\!|\cdot|\!) : \mathrm{AExp} \times \mathbb{S} \to \mathbb{Z}$. This semantic function $(\!|\cdot|\!)$ defines the standard inductive evaluation of arithmetic expressions:

$$\begin{aligned}
(\!|v|\!)s &\triangleq v \\
(\!|x_i|\!)s &\triangleq s(x_i) \\
(\!|a + a'|\!)s &\triangleq (\!|a|\!)s + (\!|a'|\!)s \\
(\!|a - a'|\!)s &\triangleq (\!|a|\!)s - (\!|a'|\!)s \\
(\!|a * a'|\!)s &\triangleq (\!|a|\!)s * (\!|a'|\!)s
\end{aligned}$$

Analogously, the semantics of Boolean expressions is given by the function $(\!|\cdot|\!) : \mathrm{BExp} \times \mathbb{S} \to \{\mathbf{t}, \mathbf{f}\}$ whose definition is straightforward and therefore omitted.

The collecting (or strongest postcondition) semantics of arithmetic expressions is modeled by the function $[\![a]\!] : \wp(\mathbb{S}) \to \wp(\mathbb{Z})$ defined by

$$[\![a]\!]S \triangleq \{(\!|a|\!)s \in \mathbb{Z} \mid s \in S\}$$

which collects all the possible semantic evaluations of $a \in \mathrm{AExp}$ for stores ranging in $S \in \wp(\mathbb{S})$. Also, the collecting semantics of Boolean expressions is the function $[\![b]\!] : \wp(\mathbb{S}) \to \wp(\mathbb{S})$ defined by

$$
\begin{aligned}
[\![\mathbf{skip}]\!]S &\triangleq S \\
[\![x_i := a]\!]S &\triangleq \{s[x_i \mapsto (\!|a|\!)s] \mid s \in S\} \\
[\![C_1; C_2]\!]S &\triangleq [\![C_2]\!][\![C_1]\!]S \\
[\![\mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2]\!]S &\triangleq [\![C_1]\!][\![b]\!]S \cup [\![C_2]\!][\![\neg b]\!]S \\
[\![\mathbf{while}\ b\ \mathbf{do}\ C]\!]S &\triangleq [\![\neg b]\!]\big(\mathsf{lfp}(\lambda T.\ S \cup [\![C]\!][\![b]\!]T)\big)
\end{aligned}
$$

Figure 2.2: Collecting denotational semantics of Prog

$$[\![b]\!]S \triangleq \{s \in S \mid (\!|b|\!)s = \mathbf{t}\}$$

so that $[\![b]\!]S \subseteq S$ filters the stores of S which make b true.

The collecting denotational program semantics is given by the function $[\![C]\!] : \wp(\mathbb{S}) \to \wp(\mathbb{S})$ and it is defined in Fig. 2.2. This is the standard predicate transformer semantics (also called strongest postcondition semantics) since $[\![P]\!]S \in \wp(\mathbb{S})$ turns out to be the strongest store predicate for the store precondition $S \in \wp(\mathbb{S})$. The terminology "collecting semantics" comes from the fact that for all $C \in \mathsf{Prog}$, $[\![C]\!] : \wp(\mathbb{S}) \to \wp(\mathbb{S})$ is an additive function on the complete lattice $\langle \wp(\mathbb{S}), \subseteq \rangle$, so that $[\![C]\!]S = \cup_{s \in S}\ [\![C]\!]\{s\}$ holds. Let $P \in \mathsf{Prog}$ be some while-program. In this case $\lambda s \in \mathbb{S}\,.[\![P]\!]\{s\}$ is the partial recursive function computed by P, where $[\![P]\!]\{s\} = \varnothing$ means non-termination of program P when evaluated in the store s. Conversely, when a program terminates on a store s we have $[\![P]\!]\{s\} = \{s'\}$ for a suitable store s'. Note that, when P does not contain any assignment to a variable x_i, if $[\![P]\!]\{s\} = \{s'\}$ then $s'(x_i) = s(x_i)$. When $[\![P]\!]$ is applied to a singleton $\{s\} \in \wp(\mathbb{S})$, we use the simpler notation $[\![P]\!]s$ in place of $[\![P]\!]\{s\}$. We will often abuse notation and represent with $[\![P]\!]$ both the above mentioned collecting semantics (i.e., a total function from set of stores to set of stores) and the ordinary denotational semantics of P (i.e., a partial function $\lambda s.[\![P]\!]\{s\}$ from stores to stores where $\varnothing$ corresponds to non-termination). The collecting semantics is typically used as reference semantics for designing static program analyses, for example, the definition in Figure 2.2 can be found in [59].

Example 2.1. Consider the following program R:

$$
\begin{aligned}
&x := 9; \\
&\mathbf{while}\ x > 1\ \mathbf{do} \\
&\qquad x := x - 2
\end{aligned}
$$

It is easy to observe that this program ends with output $\langle x \mapsto 1 \rangle$. Obviously, this result is also achieved by the collecting semantics of R by considering the

set of stores over one variable x, that is $\mathbb{Z}$. Let $S \triangleq \{\langle x \mapsto 9\rangle\} = [\![x := 9]\!]\mathbb{Z}$. The Kleene iterates of

$$\lambda T.\ S \cup [\![x := x - 2]\!][\![x > 1]\!]T$$

in $\wp(\mathbb{Z})$ converge in a finite number of steps:

$$\begin{aligned}
\varnothing &\Rightarrow \{\langle x \mapsto 9\rangle\} \\
&\Rightarrow \{\langle x \mapsto 9\rangle, \langle x \mapsto 7\rangle\} \\
&\Rightarrow \{\langle x \mapsto 9\rangle, \langle x \mapsto 7\rangle, \langle x \mapsto 5\rangle\} \\
&\Rightarrow \{\langle x \mapsto 9\rangle, \langle x \mapsto 7\rangle, \langle x \mapsto 5\rangle, \langle x \mapsto 3\rangle\} \\
&\Rightarrow \{\langle x \mapsto 9\rangle, \langle x \mapsto 7\rangle, \langle x \mapsto 5\rangle, \langle x \mapsto 3\rangle, \langle x \mapsto 1\rangle\}.
\end{aligned}$$

Therefore, the collecting semantics of program R over any set of stores $S \in \wp(\mathbb{Z})$ is $[\![R]\!]S = \{\langle x \mapsto 1\rangle\}$. ■

2.3 Elements of Computability

Let us recall some basic notions in computability theory (the reader is referred to standard textbooks such as [39, 62, 68, 70] for a comprehensive coverage).

Partial recursive functions. We will consider n-ary partial recursive functions (or Turing machines) over infinite denumerable domains such as $\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{S} \triangleq \cup_{n\in\mathbb{N}} \mathbb{Z}^n$. Let us remark that Prog is a deterministic language and this is modeled by the following property (whose easy proof is omitted) of the collecting denotational semantics defined in Figure 2.2 : for all $P \in \mathsf{Prog}$, $s_{\text{in}} \in \mathbb{S}$, if $[\![P]\!]s_{\text{in}} \neq \varnothing$ then there exists a unique $s_{\text{out}} \in \mathbb{S}$ such that $[\![P]\!]s_{\text{in}} = \{s_{\text{out}}\}$. If $[\![P]\!]s_{\text{in}} \neq \varnothing$ then $[\![P]\!]s_{\text{in}}$ will denote this unique output store $s_{\text{out}} \in \mathbb{S}$. For all programs $P \in \mathsf{Prog}$, this allows us to define the *partial recursive function* $\varphi_P : \mathbb{S} \nrightarrow \mathbb{S}$ *computed by* P as follows:

$$\varphi_P(s) \triangleq \begin{cases} [\![P]\!]s & \text{if } [\![P]\!]s \in \mathbb{S}, \\ \uparrow & \text{otherwise} \end{cases}$$

where $\uparrow$ denotes that φ_P is undefined (e.g., the execution of P does not terminate). φ_P is well-defined because Prog is deterministic.

Example 2.2. Consider the program R in Example 2.1 and the corresponding recursive function φ_R computed by R. Then φ_R is a totally defined function over $\mathbb{Z}$, indeed, for all $i \in \mathbb{Z}$, $\varphi_R(i) = 1$.

Let us define the following program Z:

if $x > 0$ **then** $x := 0$
else while true do skip

then, clearly, φ_Z is a partial recursive function because for all $i \in \mathbb{Z}_{>0}$, $\varphi_Z(i) = 0$ while for all $j \in \mathbb{Z}_{\leq 0}$, $\varphi_Z(j) = \uparrow$. ■

Recursively enumerable sets. Prog is Turing complete, so that a set of stores $S \subseteq \mathbb{S}$ is *recursively enumerable* (r.e. for short) if there exists $P \in \mathsf{Prog}$ such that $S = \mathit{dom}(\varphi_P)$ or, equivalently, $S = \mathit{range}(\varphi_P)$, while $S \subseteq \mathbb{S}$ is *recursive* when both S and $\overline{S}$ are recursively enumerable [70]. A r.e. set is said to be *strictly* r.e. if it is r.e. but not recursive. We use the following notations: given $P \in \mathsf{Prog}$, $W_P \triangleq \mathit{dom}(\varphi_P) \in \wp(\mathbb{S})$, and, in turn, $\wp^{\mathrm{re}}(\mathbb{S}) \triangleq \{W_P \in \wp(\mathbb{S}) \mid P \in \mathsf{Prog}\}$. It is known [70] that $\langle \wp^{\mathrm{re}}(\mathbb{S}), \subseteq \rangle$ is a distributive lattice with $\varnothing$ and $\mathbb{S}$ as, respectively, bottom and top elements and that the set of recursive sets $\wp^{\mathrm{rec}}(\mathbb{S}) \triangleq \{S \in \wp(\mathbb{S}) \mid S, \overline{S} \in \wp^{\mathrm{re}}(\mathbb{S})\}$ defines a Boolean algebra $\langle \wp^{\mathrm{rec}}(\mathbb{S}), \subseteq, \cup, \cap, \neg, \mathbb{S}, \varnothing \rangle$. Both $\wp^{\mathrm{re}}(\mathbb{S})$ and $\wp^{\mathrm{rec}}(\mathbb{S})$ are denumerable and $\wp^{\mathrm{rec}}(\mathbb{S}) \subset \wp^{\mathrm{re}}(\mathbb{S}) \subset \wp(\mathbb{S})$.

Despite the definition of collecting semantics applies to all subsets of $\mathbb{S}$, in the following we consider only sets of stores $S \in \wp(\mathbb{S})$ such that (i) S is r.e., namely, $S \in \wp^{\mathrm{re}}(\mathbb{S})$, and (ii) S predicates over a finite set of variables, namely $|\mathit{vars}(S)| < \omega$, as is always the case in abstract interpretation. This still allows any variable $x_i \in |\mathit{vars}(S)|$ to be assigned infinitely many different values by the stores in a set $S \in \wp^{\mathrm{re}}(\mathbb{S})$. Because any input set of stores must have a constructive computable way for building it and programs always manipulate a finite set of variables, the concrete collecting semantics of arithmetic, boolean and command expressions defined in the previous section, can be seen as restricted to r.e. variable finite sets of stores only. We highlight the previous hypothesis in the following:

Assumption 1. *In the following, unless specified, we consider as inputs to semantic functions only sets of stores that are r.e. and that predicate over a finite set of variables.*

This is an acceptable assumption since our focus is on recursive-theoretic properties of false alarm removal and injection, therefore we consider here the simplest possible Turing complete language Prog where programs manipulate a finite set of variables. Richer languages manipulating an unbounded number of variables, e.g., by recursion, can be considered at the price of complicating the model and replacing variable finiteness with abstract domains defined as functions from natural numbers $n \in \mathbb{N}$ to Galois connections on a concrete domain with n variables (e.g., see [73]).

Extensional properties. Since $[\![P]\!]$ is an additive function, it turns out that for all $S \in \wp^{\mathrm{re}}(\mathbb{S})$, $[\![P]\!]S = \{\varphi_P(s) \in \mathbb{S} \mid s \in S \wedge \varphi_P(s)\downarrow\}$. We will consider

properties of programs, i.e., sets in $\wp(\mathsf{Prog})$. A program property $\Pi \in \wp(\mathsf{Prog})$ is *extensional* if whenever $P \in \Pi$, $Q \in \mathsf{Prog}$ and $\varphi_P \cong \varphi_Q$ then $Q \in \Pi$. Any property about computable functions induces an extensional property of programs, namely those programs P whose computed function φ_P satisfies the given property. In this case Rice's Theorem [67] holds: a property of partial recursive functions (i.e., an extensional property of programs) is recursive if and only if it is trivial (i.e., satisfied by all partial recursive functions or by none).

Example 2.3. Let us define the following set of programs:

$$\mathrm{Lines}_{\geq 100} \triangleq \{P \in \mathsf{Prog} \mid |P| \geq 100\} \subset \wp(\mathsf{Prog})$$

such that $|P|$ returns the number of lines of code of program P. Then all programs in $\mathrm{Lines}_{\geq 100}$ hold the property of having at least 100 lines of instructions code. This programs property is an intensional set because it considers only how programs are written, i.e., their syntax, instead of their semantics. Conversely, consider the following set:

$$\mathrm{Out}_{id} \triangleq \{P \in \mathsf{Prog} \mid \forall s \in \mathbb{S}\,.\, \varphi_P(s) = s\} \subset \wp(\mathsf{Prog})$$

which is constituted by all programs that terminate with an output state that corresponds to the input state. This property is extensional intuitively because it "speaks" about the output of all programs in Out_{id}, therefore, it considers the programs semantics only. ■

Specializers and Interpreters. By Gödel numbering, we can associate a unique natural number with any program in Prog and conversely a unique program with any natural number by a pair of total recursive functions which forms a bijection: $\mathfrak{g} : \mathsf{Prog} \to \mathbb{N}$ and $\mathfrak{g}^{-1} : \mathbb{N} \to \mathsf{Prog}$. Since Prog is a Turing complete language, we have that $\mathscr{I} \triangleq \lambda n \in \mathbb{N}.\ \varphi_{\mathfrak{g}^{-1}(n)}$ provides an acceptable numbering for all partial recursive functions, called standard enumeration. By assuming this standard enumeration $\mathscr{I}$, in the following we denote either by φ_n the partial recursive function corresponding to the program $\mathfrak{g}^{-1}(n)$ or directly by φ_P. It is known that if $\mathscr{I}$ is an acceptable numbering system of partial recursive functions, then there exist two indexes $u, t \in \mathbb{N}$ of total recursive functions $\varphi_u, \varphi_t : \mathbb{N} \times \mathbb{S} \to \mathbb{N}$ such that:

- $\varphi_u(n, s) = \varphi_n(s)$, that is, u is the index of a universal Turing machine for partial recursive functions in $\mathbb{S} \nrightarrow \mathbb{S}$, while
- if $z \triangleq \langle x_1, ..., x_i, ..., x_j \rangle$, $x \triangleq \langle x_1, ..., x_i \rangle$ and $y \triangleq \langle x_{i+1}, ..., x_j \rangle$ with $1 < i \leq j$, then $\varphi_{\varphi_t(n,x)}(y) = \varphi_n(z)$, that is, t is the index of a Turing machine that specializes the program with index n with input x.

This means that there exists programs $\mathsf{Interp}, \mathsf{Spec} \in \mathsf{Prog}$, respectively, the *interpreter* and the *specialiser*, that are an implementation in Prog of the total recursive functions, respectively, φ_u and φ_t. Interp is called an *enumeration* and Spec is called a *parametrisation* or the s-m-n theorem.

Creative and Productive sets. We have seen that not all elements in $\wp(\mathbb{S})$ are recursively enumerable. A set $S \in \wp(\mathbb{S})$ is *creative* if it is r.e. and its complement $\overline{S} = \mathbb{S} \setminus S$ is *productive*, i.e., there exists a total recursive function $f : \mathsf{Prog} \to \mathbb{S}$ such that

$$\forall P \in \mathsf{Prog}\,.\, W_P \subseteq \overline{S} \implies f(P) \in \overline{S} \setminus W_P.$$

It is clear that all productive sets are not r.e. while creative sets are strictly r.e., thus not recursive. It is also known that, while the complement of a creative set is always productive, the complement of a productive set may be productive [61, 65]. This is the case, e.g., for the set $H \triangleq \{P \in \mathsf{Prog} \mid W_P \text{ is recursive}\}$ [35]. For all $A \in \wp^{\mathrm{re}}(\mathbb{S})$ and $S \in \wp(\mathbb{S})$, we denote with $A \preceq_f S$ the *many-to-one reducibility*, where $f : \mathbb{S} \to \mathbb{S}$ is a total recursive function such that for all $s \in \mathbb{S}$, $s \in A \;\Leftrightarrow\; f(s) \in S$. In some books, the reduction $\preceq_f$ is also denoted by $\preceq_m$, e.g., in [68]. Let us observe that this definition allows reductions between sets of tuples of different sizes. Moreover, recall that if S is productive and $S \preceq_f X$ then also X is productive. It is also known that creative sets are *complete* in $\wp^{\mathrm{re}}(\mathbb{S})$, i.e., $S \in \wp(\mathbb{S})$ is creative iff it is r.e. and for every $A \in \wp^{\mathrm{re}}(\mathbb{S})$, $A \preceq_f S$. Creative sets play a special role (maximal) in the class of r.e. sets. The overall hierarchy is shown in Figure 2.3.

Example 2.4. A classical example of, respectively, creative set (therefore r.e.) and productive set (therefore non-r.e.) are the representations in Prog of the halting problem of Turing machine (cf. [68]):

$$K \triangleq \{P \in \mathsf{Prog} \mid \mathfrak{g}(P) \in W_P\}$$

and its complement set $\overline{K}$, where here, with a slight abuse of notation, W_P is the set of indices corresponding to each individual input of $dom(\varphi_P)$, namely $W_P \triangleq \{\mathfrak{g}_{\mathbb{S}}(s) \mid s \in \mathbb{S} \wedge s \in dom(\varphi_P)\}$ with $\mathfrak{g}_{\mathbb{S}} : \mathbb{S} \to \mathbb{N}$ the Gödel number associated to each store in $\mathbb{S}$. ■

Arithmetical hierarchy. The *Kleene arithmetical hierarchy* [51] is particularly important to compare properties of programs. We denote by $\Sigma_0 = \Pi_0 = \Delta_0$ the set of all recursive sets, i.e., $\Sigma_0 = \Pi_0 = \Delta_0 \triangleq \wp^{\mathrm{rec}}(\mathbb{S})$. For $n \geq 1$ we define when a set $A \in \wp^{\mathrm{rec}}(\mathbb{S})$ is arithmetical as follows:

- $A \in \Sigma_n$ if there exists a recursive predicate $R(x, y_1, \ldots, y_n) \subseteq \mathbb{S}$ such that

$$x \in A \;\Leftrightarrow\; \exists y_1.\, \forall y_2.\, \ldots Q y_n.\, R(x, y_1, \ldots, y_n)$$

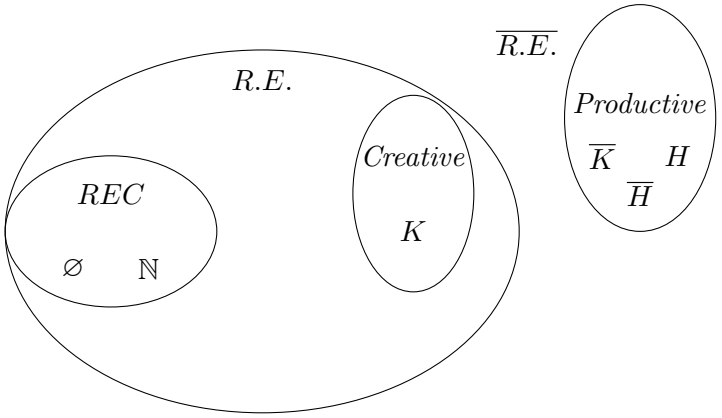


Figure 2.3: Hierarchy of recursive sets

where $Q = \exists$ if n is odd, while $Q = \forall$ if n is even.

- $A \in \Pi_n$ if there exists a recursive predicate $R(x, y_1, \ldots, y_n) \subseteq \mathbb{S}$ such that

$$x \in A \;\Leftrightarrow\; \forall y_1.\, \exists y_2.\, \ldots Q y_n.\, R(x, y_1, \ldots, y_n)$$

where $Q = \forall$ if n is odd, while $Q = \exists$ if n is even.

- $A \in \Delta_n$ if $A \in \Sigma_n \cap \Pi_n$.

Because redundant quantifiers can be added to any formula, once a formula is assigned the classification Σ_n or Π_n it will be assigned the classifications Σ_r and Π_r for every $r > n$. The most important classification assigned to a formula is thus the one with the least n, because this is enough to determine all the other classifications. The following results are known to be true [68] :

(i) $A \in \Sigma_n \;\Leftrightarrow\; \overline{A} \in \Pi_n$;

(ii) $A \in \Sigma_n \cup \Pi_n \;\Rightarrow\; \forall m > n.\, A \in \Delta_m = \Sigma_m \cap \Pi_m$;

(iii) $B \preceq_f A \;\wedge\; A \in \Sigma_n \;\Rightarrow\; B \in \Sigma_n$;

(iv) $R \in \Sigma_{n>0} \;\wedge\; A \triangleq \{x \mid \exists y.\, R(x, y)\} \;\Rightarrow\; A \in \Sigma_n$.

Example 2.5. The set Σ_1 corresponds exactly to the set of all r.e. sets. In fact, those sets can be defined by a recursive predicate of the form

$$\exists y_1 \ldots \exists y_n.\, R(y_1, \ldots, y_n, x).$$

The set $H \triangleq \{P \in \mathsf{Prog} \mid W_P \text{ is recursive}\}$ of all programs whose domain set forms a recursive set, can be characterized by the following predicate:

$$P \in H \;\Leftrightarrow\; \forall s \in \mathbb{S}\,.\, \exists i \in \mathbb{N}.\, \varphi_P(s)\!\downarrow \text{ in } i \text{ steps.}$$

Note that the predicate "$\varphi_P(s)\downarrow$ in i steps" is computable, therefore we can conlude that $H \in \Pi_2$, i.e., it is non-r.e. . ■

2.4 Formal Language Theory Basics

In this section we recall some notions of formal language theory (for a comprehensive coverage, the reader is referred to standard textbooks such as [39, 48]).

Alphabet and languages. Let Σ be an alphabet (that is, a finite nonempty set of symbols). Words are finite sequences of symbols where ϵ denotes the empty sequence. The length of a word u, denoted $|u|$, is the number of letters it is composed of. For every alphabet, there is only one word of length 0, namely ϵ. A *formal language* (language briefly) $L \subseteq \Sigma^*$ is a set of words, where Σ^* is the set of all words over the alphabet Σ and $*$ denotes the Kleene closure operation such that $\Sigma^* \triangleq \bigcup_{i \geq 0} \Sigma^i$. Concatenation in Σ^* is simply denoted by juxtaposition, both for concatenating words uv, languages $L_1 L_2$ and words with languages such as uLv. The positive closure of L is denoted L^+ and it is defined as $L^+ \triangleq LL^*$.

Regular languages. The class of *regular languages*, denoted REG, is the collection of all languages defined recursively as follows:

(i) the empty language $\varnothing$ is a regular language;

(ii) for each symbol $a \in \Sigma$, the singleton language $\{a\}$ is a regular language;

(iii) if R is a regular language, R^* is a regular language (due to this, the empty string language $\{\epsilon\}$ is also regular);

(iv) if L and R are regular languages, then $L \cup R$ (union) and LR (concatenation) are regular languages;

(v) no other languages over Σ are regular.

Note that all finite languages are regular. Regular languages can be equivalently defined as the languages recognized by nondeterministic finite automaton (NFA).

Example 2.6. The language consisting of all strings over the alphabet $\Sigma \triangleq \{a, b\}$ which contain an even number of a's, is a regular language.

A classic example of a language that is not regular is the set of strings $\{a^n b^n \mid n \geq 0\}$. Intuitively, it cannot be recognized with a finite automaton, since a finite automaton has finite memory and it cannot remember the exact number of a's. ■

Context-free languages. The class of *context-free languages*, denoted CF, is the collection of all languages that can be generated by a context-free (CF) grammar. A CF grammar is a quadruple $G \triangleq (N, T, P, S)$ such that:

(i) N is a finite set of nonterminal symbols (also called variables);

(ii) T is a finite set of terminal symbols such that $N \cap T = \varnothing$;

(iii) P is a finite set of productions having the form $A \to \beta$, where $A \in N$ and $\beta \in (N \cup T)^*$; multiple productions of the same nonterminal are written $A \to \beta_1 \mid \cdots \mid \beta_n$;

(iv) $S \in N$ is the start symbol.

From now on the set of non-terminals N will range over superscript symbols such as $A, B, C, \ldots$, while the set of terminals T on symbols a, b, c, d, e. A derivation in G is a sequence of strings $\beta_1, \beta_2, \ldots, \beta_n$, where β_{i+1} is derived from β_i by the application of one production in P, written $\beta_i \to_G \beta_{i+1}$. The subscript G is dropped whenever G is clearly understood. Let $\to_G^*$ be the reflexive and transitive closure of $\to_G$ defined as usual: $\beta \to_G^0 \beta$ for $n \geq 0$, $\alpha \to_G^{n+1} \gamma$ if $\exists \beta : \alpha \to_G^n \beta$ and $\beta \to_G \gamma$; $\alpha \to_G^* \beta$ iff $\alpha \to_G^i \beta$ for some $i \geq 0$. The language $\mathscr{L}(G)$ recognized by a CF grammar G is

$$\mathscr{L}(G) \triangleq \{w \in T^* \mid S \to_G^* w\}.$$

The set of all CF languages is identical to the set of languages accepted by pushdown automata, which correspond to NFA augmented with an auxiliary stack memory.

Example 2.7. The syntax of while-programs defined in Figure 2.1 is a CFG whose generated language is exactly Prog.

The non-regular language $\{a^n b^n \mid n \geq 0\}$ can be generated by the following CFG: $S \to aSb \mid \epsilon$.

A classical example of non-CF language is $\{a^n b^n c^n \mid n \geq 1\}$. ■

Context-sensitive languages. The class of *context-sensitive languages*, denoted CS, is the collection of all languages that can be generated by a context-sensitive (CS) grammar. CS grammars are similar to CF grammars but all rules in P are of the form $\alpha A \beta \to \alpha \gamma \beta$, with $A \in N$, $\alpha, \beta \in (N \cup T)^*$ and $\gamma \in (N \cup T)^+$. The name context-sensitive is explained by the α and β that form the context of A and determine whether A can be replaced with γ or not. This is different from a context-free grammar where the context of a nonterminal is not taken into consideration. Indeed, every production of a CF grammar is of the form $V \to w$ where V is a single nonterminal symbol, and w is a string of terminals and/or nonterminals; w can be empty. Note that each CF grammar can be written as a

CS grammar, thus generating all CF languages. The language $\mathscr{L}(G)$ recognized by a CS grammar G is $\mathscr{L}(G) \triangleq \{w \in T^* \mid S \to_G^* w\}$. A language can be described by a CS grammar if and only if it is accepted by some linear bounded automaton (LBA) which is similar to a nondeterministic Turing machine but the tape has not unbounded length.

Example 2.8. The non-CF language $\{a^n b^n c^n \mid n \geq 1\}$ can be generated by the following CS grammar:

$$\begin{array}{lll} S \to aSBC \mid aBC & aB \to ab & bC \to bc \\ CB \to BC & bB \to bb & cC \to cc \end{array}$$

■

It is worth remarking that if a language L is in REG, CF or CS, then L is a recursive set. The following is known as the *Chomsky's hierarchy*:

$$REG \subset CF \subset CS \subset \wp^{\mathrm{rec}}(\Sigma^*) \subset \wp^{\mathrm{re}}(\Sigma^*) \subset \wp(\Sigma^*).$$

Chapter 3

ABSTRACT INTERPRETATION THEORY

In this chapter, we recall some basic concepts about the abstract interpretation framework, which plays a central role in the results presented in Chapters 4 and 5. Moreover, we start defining some essential notions, like store abstractions and recursive abstract domains of stores, which will be considered in the context of static program analysis. We start with Section 3.1 by defining the general concepts of abstract domains, abstractions and closures. Section 3.2 defines when an approximation of a concrete operator is correct and complete. In Section 3.3 we define store abstractions which will be considered in Chapter 4. Then, in Section 3.4 we recall the abstract denotational semantics of while programs. In Section 3.5, we give the definition of recursive and trivial abstract domains of stores. Lastly, in Section 3.6 we recall the definition of completeness class of programs, whose weakening will be the main focus in the next chapter.

3.1 Abstract Domains

Abstract interpretation [26,27] is a theory of sound approximation of the semantics of computer programs. It can be viewed as a partial execution of a computer program which gains information about its semantics without performing all the calculations. The underlying idea of program analysis by abstract interpretation is strikingly simple: extracting a program property means approximating its semantics. The standard abstract interpretation framework [22, 26, 27] is based on the correspondence between a domain of concrete or exact properties and a domain of abstract or approximate properties. Abstract domains (also called ab-

stractions) are specified by Galois connections/insertions (GCs/GIs for short). In program analysis, concrete and abstract domains are assumed to be complete lattices, resp. $\langle C, \leq_C \rangle$ and $\langle A, \leq_A \rangle$, which are related by abstraction and concretization maps $\alpha_A : C \to A$ and $\gamma_A : A \to C$ that give rise to a GC $(\alpha_A, C, A, \gamma_A)$, that is, for all $a \in A$ and $c \in C$:

$$\alpha_A(c) \leq_A a \Leftrightarrow c \leq_C \gamma_A(a)$$

where here and in the following we use the subscript to functions α_A and γ_A in order to emphasize the abstract domain A considered. A GC is a GI when $\alpha_A \circ \gamma_A = \lambda x.x$. Let us recall some basic properties of a GC $(\alpha_A, C, A, \gamma_A)$:

(1) α_A is additive and γ_A is co-additive;

(2) $\gamma_A \circ \alpha_A : C \to C$ is a closure operator, namely, it is a monotone, idempotent and increasing function;

(3) if $\rho : C \to C$ is a closure operator then $(\rho, C, \rho(C), \lambda x.x)$ is a GI.

Assumption 2. *In the rest of the thesis, we will only consider Galois insertion-based abstract interpretations.*

If $\alpha_A : C \to A$ is an additive function, then it induces a GC $(\alpha_A, C, A, \alpha_A^+)$ where the concretization $\alpha_A^+ : A \to C$ is defined as right-adjoint of α_A, i.e., $\alpha_A^+(a) \triangleq \vee_C \{c \in C \mid \alpha_A(c) \leq_A a\}$. Dually, if $\gamma_A : C \to A$ is a co-additive function then $(\gamma_A^-, C, A, \gamma_A)$ is a GC where $\gamma_A^- \triangleq \lambda c. \wedge_A \{a \in A \mid c \leq_C \gamma_A(a)\}$ is the left-adjoint of γ_A.

We use $\mathscr{A}(C)$ to denote all the possible abstractions of a concrete domain C, where $A \in \mathscr{A}(C)$ means that A is an abstract domain of C defined by some GI which is left unspecified. An abstract domain $A \in \mathscr{A}(C)$ is called *strict* if $\gamma_A(\bot_A) = \bot_C$. We say that an element $c \in C$ is *exactly represented* in A if $\gamma_A(\alpha_A(c)) = c$. If $A_1, A_2 \in \mathscr{A}(C)$ then A_1 is equivalent to A_2, denoted by $A_1 \sim_{\mathscr{A}(C)} A_2$, when $\gamma_{A_1}(A_1) = \gamma_{A_2}(A_2)$. The quotient $\mathscr{A}(C)_{/\sim_{\mathscr{A}(C)}}$ is called the lattice of abstractions because it turns out to be a complete lattice w.r.t. the relative precision ordering: $A_1 \leq_{\mathscr{A}(C)} A_2$ iff for all $c \in C$, $\gamma_{A_1}(\alpha_{A_1}(c)) \leq_C \gamma_{A_2}(\alpha_{A_2}(c))$. Thus, $A_1 \leq_{\mathscr{A}(C)} A_2$ means that A_1 is a more precise abstraction than A_2, or, equivalently, that A_2 abstracts A_1. The following are abstract domain examples of $\mathscr{A}(\wp^{\mathrm{re}}(\mathbb{Z}))$.

Example 3.1. The Sign abstract domain $\mathsf{Sign} \triangleq \{\mathbb{Z}, -, 0, +, \varnothing\}$ depicted in Figure 3.1, for sign analysis of integer variables, is a straightforward non-relational abstraction of $\langle \wp^{\mathrm{re}}(\mathbb{Z}), \subseteq \rangle$ [21], where the order relation $\leq_{\mathsf{Sign}}$ is defined as $\varnothing <_{\mathsf{Sign}} 0 <_{\mathsf{Sign}} - <_{\mathsf{Sign}} \mathbb{Z}$ and $\varnothing <_{\mathsf{Sign}} 0 <_{\mathsf{Sign}} + <_{\mathsf{Sign}} \mathbb{Z}$. The abstraction map $\alpha_{\mathsf{Sign}} : \wp^{\mathrm{re}}(\mathbb{Z}) \to \mathsf{Sign}$ is defined by:

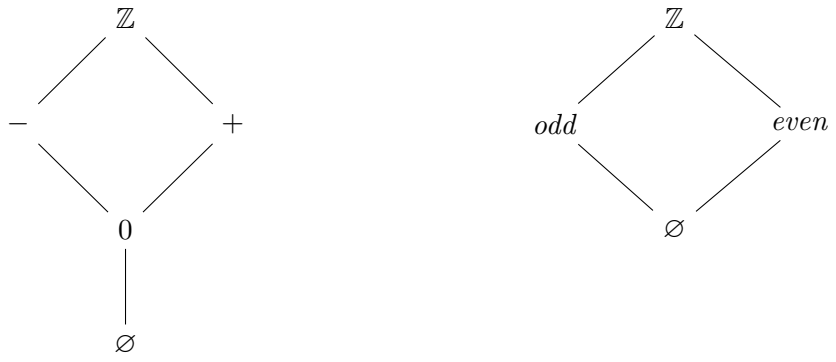


Figure 3.1: Sign abstract domain (on the left) and Parity abstract domain (on the right)

$$\alpha_{\mathsf{Sign}}(X) \triangleq \begin{cases} \varnothing & \text{if } X = \varnothing, \\ 0 & \text{if } X = \{0\}, \\ + & \text{if } \forall x \in X.\ x \geq 0, \\ - & \text{if } \forall x \in X.\ x \leq 0, \\ \mathbb{Z} & \text{otherwise.} \end{cases}$$

■

Example 3.2. The Parity abstract domain is defined as $\mathsf{Parity} \triangleq \{\mathbb{Z}, \mathit{even}, \mathit{odd}, \varnothing\}$ and shown in Figure 3.1. It is a straightforward non-relational abstraction of $\langle \wp^{\mathrm{re}}(\mathbb{Z}), \subseteq \rangle$ [21] for parity analysis of integer variables. The order relation is defined as: $\varnothing <_{\mathsf{Parity}} \mathit{even} <_{\mathsf{Parity}} \mathbb{Z}$ and $\varnothing <_{\mathsf{Parity}} \mathit{odd} <_{\mathsf{Parity}} \mathbb{Z}$. The abstraction map is defined by the function $\alpha_{\mathsf{Parity}} : \wp^{\mathrm{re}}(\mathbb{Z}) \to \mathsf{Parity}$ as:

$$\alpha_{\mathsf{Parity}}(X) \triangleq \begin{cases} \varnothing & \text{if } X = \varnothing, \\ \mathit{even} & \text{if } \forall x \in X.\ x \bmod 2 = 0, \\ \mathit{odd} & \text{if } \forall x \in X.\ x \bmod 2 \neq 0, \\ \mathbb{Z} & \text{otherwise} \end{cases}$$

where mod is the integer modulo operation. ■

Example 3.3. The interval abstraction Int [21] shown in Figure 3.2, is a widely used non-relational abstraction since it is efficient and yet able to give useful information to prove, e.g., the absence of arithmetic overflows or out-of-bounds array accesses. Let $\mathbb{Z}^* \triangleq \mathbb{Z} \cup \{-\infty, +\infty\}$ and assume that the standard ordering $\leq$ on $\mathbb{Z}$ is extended to $\mathbb{Z}^*$ in the usual way. Hence:

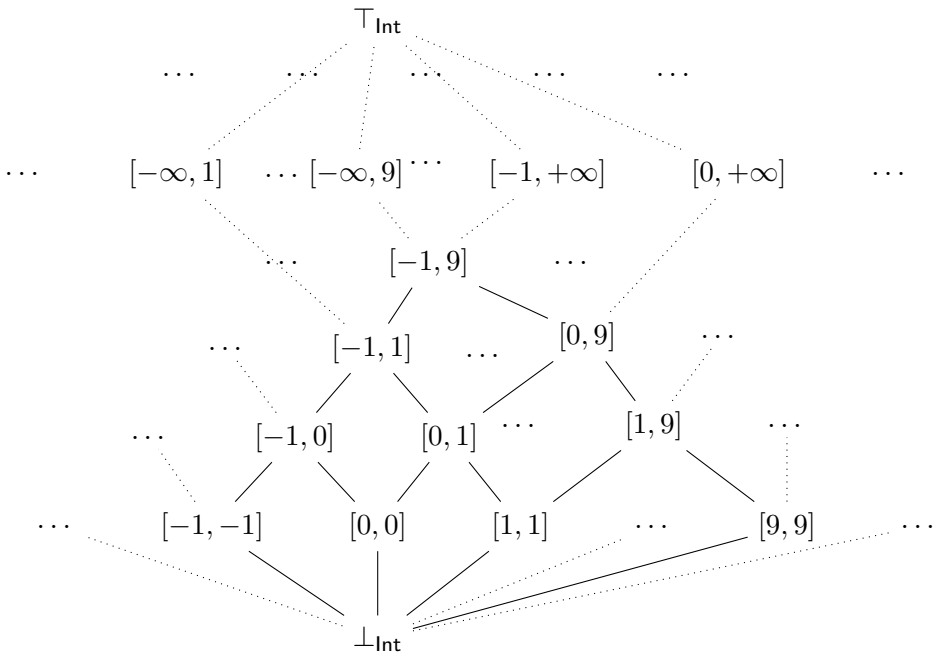


Figure 3.2: The Int abstract domain

$$\mathsf{Int} \triangleq \{[a, b] \mid a, b \in \mathbb{Z}^*,\ a \leq b\} \cup \{\bot_{\mathsf{Int}}\}$$

endowed with the standard ordering $\leq_{\mathsf{Int}}$ induced by the interval containment gives rise to a complete lattice, where $\bot_{\mathsf{Int}}$ is the bottom element and $\top_{\mathsf{Int}} \triangleq [-\infty, +\infty]$ is the top element. Then, consider the function $min : \wp^{\mathrm{re}}(\mathbb{Z}) \to \mathbb{Z}^*$ defined as follows:

$$min(X) \triangleq \begin{cases} x & \text{if } \exists x \in X.\ \forall y \in X.\ x \leq y, \\ -\infty & \text{otherwise} \end{cases}$$

while max is dually defined. The abstraction map $\alpha_{\mathsf{Int}} : \wp^{\mathrm{re}}(\mathbb{Z}) \to \mathsf{Int}$ is defined by:

$$\alpha_{\mathsf{Int}}(X) \triangleq \begin{cases} \bot_{\mathsf{Int}} & \text{if } X = \varnothing, \\ [min(X), max(X)] & \text{otherwise} \end{cases}$$

Note that α_{Int} preserves arbitrary unions in $\wp^{\mathrm{re}}(\mathbb{Z})$ and therefore gives rise to a GI, i.e., $\mathsf{Int} \in \mathscr{A}(\wp^{\mathrm{re}}(\mathbb{Z}))$. ■

Abstract domains can be equivalently formulated in terms of closure operators [27]. An *Upper Closure Operator* (uco), or simply a *closure*, $\rho \in C \to C$ on

a poset $\langle C, \leq_C \rangle$ is an operator that is monotone, idempotent and increasing. A basic property of closure operators is that each closure $\rho \in \mathrm{uco}(C)$ is uniquely determined by the set of its fixpoints, which coincides with its image $\rho(C)$, as follows: $X \subseteq C$ is the set of fixpoints of a closure operator on C iff X is a *Moore-family* of C, that is, $X = \mathscr{M}(X) \triangleq \{\wedge_C S \mid S \subseteq X\}$, where $\wedge\varnothing = \top_C \in \mathscr{M}(X)$. In this case $\rho_X \triangleq \lambda y.\ \wedge_C \{x \in X \mid y \leq_C x\}$ is the corresponding closure operator on C. Given $X \subseteq C$, $\mathscr{M}(X)$ is called the Moore-closure of X in C, that is, $\mathscr{M}(X)$ is the least (with respect to inclusion) subset of C which contains X and is a Moore-family of C. It turns out that $\langle \rho(C), \leq_C \rangle$ is a complete meet subsemilattice of C (i.e., $\wedge_C$ is its glb), but, in general, it is not a complete sublattice of C, since the lub in $\rho(C)$, defined by $\lambda Y \subseteq \rho(C).\ \rho(\vee_C Y)$, might be different from that in C. In fact, $\rho(C)$ is a complete sublattice of C iff ρ is additive. If C is a complete lattice, then $\mathrm{uco}(C)$ ordered pointwise is a complete lattice as well. It will be denoted by $\langle \mathrm{uco}(C), \sqsubseteq, \sqcup, \sqcap, \lambda x.\top_C, id \rangle$ where $id \triangleq \lambda x.x$ and for all $\rho, \eta \in \mathrm{uco}(C)$, $\{\rho_i\}_{i\in I} \subseteq \mathrm{uco}(C)$ and $x \in C$:

- $\rho \sqsubseteq \eta$ iff $\forall y \in C.\ \rho(y) \leq_C \eta(y)$ iff $\eta(C) \subseteq_C \rho(C)$;
- $(\bigsqcap_{i\in I} \rho_i)(x) = \wedge_{C\ i\in I} \rho_i(x)$;
- $(\bigsqcup_{i\in I} \rho_i)(x) = x \Leftrightarrow \forall i \in I.\ \rho_i(x) = x$;
- $\lambda.\top_C$ is the top element, whereas $\lambda x.x$ is the bottom element.

Thus, the glb in $\mathrm{uco}(C)$ is defined pointwise, while the lub of a set of closures $\{\rho_i\}_{i\in I} \subseteq \mathrm{uco}(C)$ is the closure whose set of fixpoints is given by the set-intersection $\bigcap_{i\in I \rho_i(C)}$. In the following, we will make use of the following properties: For $\rho, \eta \in uco(C)$ and $Y \subseteq_C C$

(i) $\rho(\wedge_C \rho(Y)) = \wedge_C \rho(Y)$;

(ii) $\rho(\vee_C Y) = \rho(\vee_C \rho(Y))$;

(iii) $\eta \sqsubseteq \rho \Leftrightarrow \eta \circ \rho = \rho \Leftrightarrow \rho \circ \eta = \rho$.

3.2 Correctness and Completeness

Let $f : C \to C$ be a concrete monotone function—to keep notation simple, we consider unary functions—and let $f^\sharp : A \to A$ be a corresponding monotone abstract function defined on some abstraction $A \in \mathscr{A}(C)$. Then, $f^\sharp$ is a *correct* (or *sound*) approximation of f on A when $\alpha_A \circ f \leq_A f^\sharp \circ \alpha_A$ holds (Figure 3.3). If $f^\sharp$ is correct for f then least fixpoint correctness holds, that is, $\alpha_A(\mathsf{lfp}(f)) \leq_A \mathsf{lfp}(f^\sharp)$ holds. The abstract function $f^\alpha \triangleq \alpha_A \circ f \circ \gamma_A : A \to A$ is called the *best correct approximation* (bca) of f on A, because it turns out that any abstract monotone function $f^\sharp$ is a correct approximation of f iff $f^\alpha \leq_A f^\sharp$. Hence, f^α

plays the role of the best possible correct approximation of f on the abstract domain A.

An abstract function $f^\sharp : A \to A$ is a *complete* approximation of f on A when $\alpha_A \circ f = f^\sharp \circ \alpha_A$ holds [27]. Figure 3.4 shows the general idea of completeness in abstract interpretation. Our definition of completeness corresponds to the backward completeness as defined in [43]. When $f^\sharp$ is an abstract transfer function on A used by some static program analysis, completeness intuitively encodes an optimal precision for $f^\sharp$, meaning that the abstract behavior of $f^\sharp$ on A exactly matches the abstraction in A of the concrete behaviour of f. If $f^\sharp$ is complete for f then least fixpoint completeness holds (also called fixpoint transfer), i.e., $\alpha_A(\mathsf{lfp}(f)) = \mathsf{lfp}(f^\sharp)$ holds. It turns out that completeness $\alpha_A \circ f = f^\sharp \circ \alpha_A$ holds iff $\alpha_A \circ f = \alpha_A \circ f \circ \gamma_A \circ \alpha_A = f^\alpha \circ \alpha_A$ holds. Thus, the possibility of defining a complete approximation $f^\sharp$ of f on some $A \in \mathscr{A}(C)$ only depends on the concrete function f and on the abstraction A, that is, f^α is the only possible option as complete approximation of f. Let us recall that function composition preserves completeness, that is, if f and g are complete on A then $f \circ g$ is complete on A, as shown by the following equality:

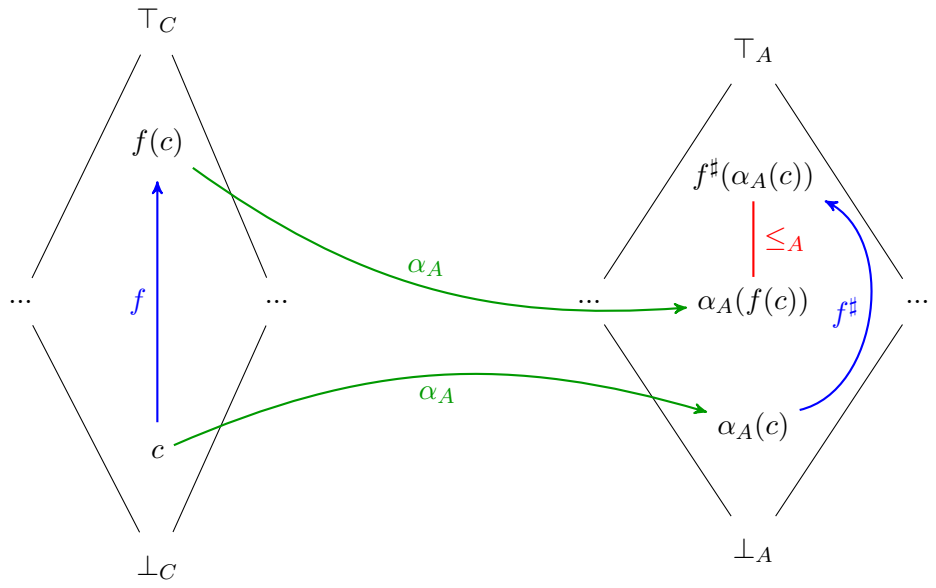
$$\begin{aligned}\alpha_A \circ f \circ g &= \alpha_A \circ f \circ \gamma_A \circ \alpha_A \circ g \\ &= \alpha_A \circ f \circ \gamma_A \circ \alpha_A \circ g \circ \gamma_A \circ \alpha_A \\ &= \alpha_A \circ f \circ g \circ \gamma_A \circ \alpha_A\end{aligned}$$

The problem of making abstract domains complete has been solved in [43]. A constructive characterization of the most abstract refinement, called complete shell, and of the most concrete simplification, called complete core, of any abstract domain $A \in \mathscr{A}(C)$, making it complete for a given continuous function $f : C \to C$, is given as a fixpoint solution of an abstract domain equation derived from f and A.

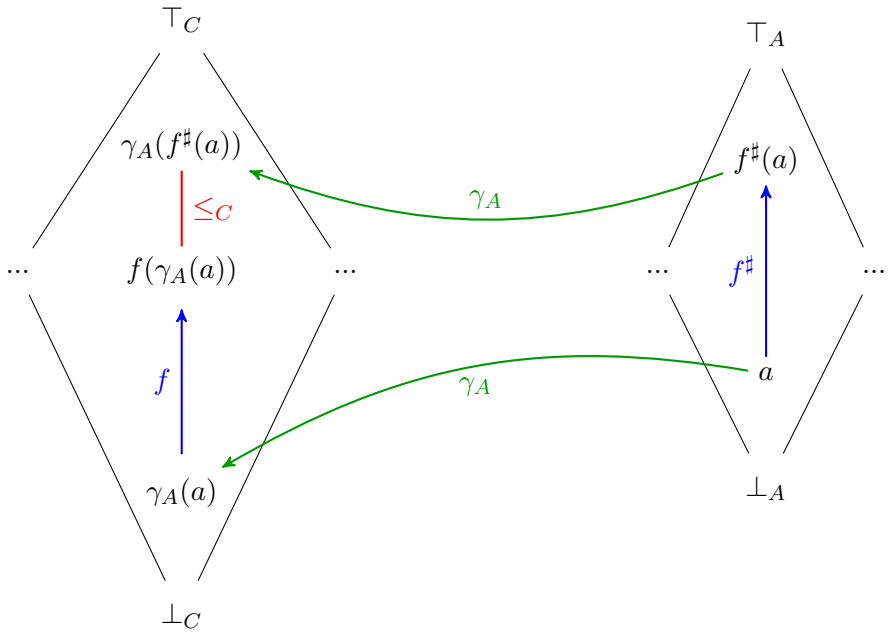
3.3 Store Abstractions

An abstraction of stores is specified by an abstraction $A \in \mathscr{A}(\wp^{\mathrm{re}}(\mathbb{S}))$ on r.e. sets of stores. This is formalized by relating abstract domains with a different number of variables through existential variable quantification, which turns out to be an abstraction. Given some $n \in \mathbb{N}$, let us define the maps $\alpha_{\exists_n} : \wp^{\mathrm{re}}(\mathbb{Z}^{n+1}) \to \wp^{\mathrm{re}}(\mathbb{Z}^n)$ and $\gamma_{\exists_n} : \wp^{\mathrm{re}}(\mathbb{Z}^n) \to \wp^{\mathrm{re}}(\mathbb{Z}^{n+1})$ as follows:

$$\begin{aligned}&\alpha_{\exists_n}(X) \triangleq \{\langle x_1, ..., x_n\rangle \in \mathbb{Z}^n \mid \langle x_1, ..., x_n, x_{n+1}\rangle \in X\},\\ &\gamma_{\exists_n}(Y) \triangleq \{\langle y_1, ..., y_n, y_{n+1}\rangle \in \mathbb{Z}^{n+1} \mid \langle y_1, ..., y_n\rangle \in Y, y_{n+1} \in \mathbb{Z}\}.\end{aligned}$$



(a) $\forall c \in C : \alpha_A(f(c)) \leq_A f^\sharp(\alpha_A(c))$



(b) $\forall a \in A : f(\gamma_A(a)) \leq_C \gamma_A(f^\sharp(a))$

Figure 3.3: Soundness

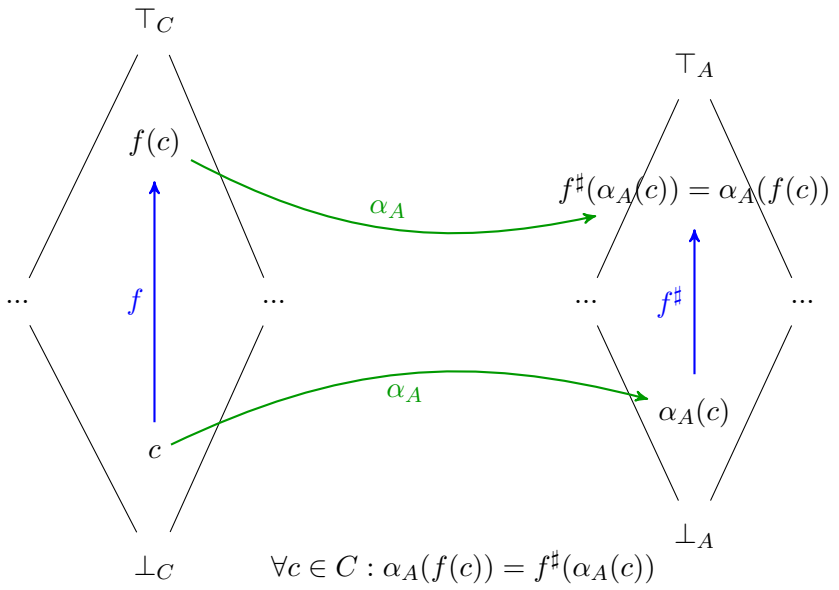


Figure 3.4: (Backward) Completeness

Observe that in the degenerate case $n = 0$ we have that $\wp^{\mathrm{re}}(\mathbb{Z}^0) = \{\varnothing, \{\epsilon\}\}$ and $\gamma_{\exists_0}(\varnothing) = \varnothing$, $\gamma_{\exists_0}(\{\epsilon\}) = \mathbb{Z}$, $\alpha_{\exists_0}(\varnothing) = \varnothing$ and $\alpha_{\exists_0}(X) = \{\epsilon\}$ if $X \neq \varnothing$.

Proposition 3.4. *Let $X \subseteq \mathbb{Z}^{n+1}$ and $Y \subseteq \mathbb{Z}^n$. If X and Y are recursive (r.e.) sets then $\alpha_{\exists_n}(X)$ and $\gamma_{\exists_n}(Y)$ are recursive (r.e.) sets.*

Proof. Immediate because $\alpha_{\exists_n}(X)$ is the range of a recursive (r.e.) set. Analogously, the recursivity (r.e.) of $\gamma_{\exists_n}(Y)$ follows from the recursivity (r.e.) of Y because for all $x \in \mathbb{Z}^n$ and $z \in \mathbb{Z}$: $\langle x, z\rangle \in \gamma_{\exists_n}(Y) \Leftrightarrow x \in Y$. □

Moreover, $(\alpha_{\exists_n}, \langle \wp^{\mathrm{re}}(\mathbb{Z}^{n+1}), \subseteq\rangle, \langle \wp^{\mathrm{re}}(\mathbb{Z}^n), \subseteq\rangle, \gamma_{\exists_n})$ is a GI for all $n \in \mathbb{N}$ which we use in our notion of store abstraction to relate abstract domains having a different number of variables.

Definition 3.5 (Store Abstraction). *A store abstraction $A \in \mathscr{A}(\wp^{\mathrm{re}}(\mathbb{Z}^n))$ is such that for all $n \in \mathbb{N}$, $\gamma_A \circ \alpha_A = \alpha_{\exists_n} \circ \gamma_A \circ \alpha_A \circ \gamma_{\exists_n}$.*

Thus, we can handle abstractions with a different number of variables: any abstraction in $\mathscr{A}(\wp^{\mathrm{re}}(\mathbb{Z}^n))$ can be seen as the bca of the corresponding abstraction in $\mathscr{A}(\wp^{\mathrm{re}}(\mathbb{Z}^{n+1}))$ w.r.t. $\langle \alpha_{\exists_n}, \gamma_{\exists_n}\rangle$. This means that, from now on, we can ignore the number of variables considered by the different abstract domains. Observe that in the degenerate case $n = 0$, since $\wp^{\mathrm{re}}(\mathbb{Z}^0) = \{\varnothing, \{\epsilon\}\}$, we have that $\mathscr{A}(\wp^{\mathrm{re}}(\mathbb{Z}^0))$ is necessarily either the identical abstraction or the top abstraction (both defined in Section 3.5).

3.4 Abstract Semantics of While Programs

The main source of imprecision in program analysis by abstract interpretation is due to the non-compositional property of abstract interpretation, namely, the composition of two best correct abstract semantics may not be the best correct abstraction of the semantics of the two components. However, we want to define the abstract semantics for a generic abstract domain of stores $A \in \mathscr{A}(\wp^{\mathrm{re}}(\mathbb{S}))$ whose loss of precision is induced only by the inductive composition of commands. Therefore, for arithmetic and boolean expressions, we consider as abstract semantics their best correct approximating semantics in A, assuming that the bca of arithmetic and boolean expressions is computable in our language Prog. Although in program analysis this is a recurrent assumption, the recursivity results presented in the next chapter do not depend on it.

Given an abstraction of stores $A \in \mathscr{A}(\wp^{\mathrm{re}}(\mathbb{S}))$ and $S^\sharp \in A$, the best correct abstract semantics on A of Boolean expressions $b \in \mathrm{BExp}$ is given by the function $[\![b]\!]^A : A \to A$ defined by

$$[\![b]\!]^A S^\sharp \triangleq \alpha_A([\![b]\!]\gamma_A(S^\sharp)).$$

A Boolean expression is defined to be complete for A if $\alpha_A([\![b]\!]S) = [\![b]\!]^A\alpha_A(S)$. The best correct abstract semantics for Arithmetic expressions $a \in \mathrm{AExp}$ is given by the function $[\![a]\!]^A : A \to A$ defined by

$$[\![a]\!]^A S^\sharp \triangleq \alpha_A([\![a]\!]\gamma_A(S^\sharp)).$$

An abstraction of stores $A \in \mathscr{A}(\wp^{\mathrm{re}}(\mathbb{S}))$ is complete for an arithmetic expression when for every set of stores $S \in \wp^{\mathrm{re}}(\mathbb{S})$: $\alpha_A([\![a]\!]S) = [\![a]\!]^A\alpha_A(S)$.
Let us remark that, in general, the computability of the best correct abstract semantics of boolean and arithmetic expressions does not hold in static program analysis by abstract interpretation which rely on the compositional abstract semantics of arithmetic and boolean expressions.

Example 3.6. Consider the interval abstract domain Int defined in Example 3.3. Given the expression $x - x \in \mathrm{AExp}$, we have the following best correct abstract evaluation on Int:

$$\begin{aligned}
[\![x - x]\!]^{\mathsf{Int}}\langle x \mapsto [1,2]\rangle &= \alpha_{\mathsf{Int}}([\![x - x]\!]\gamma_{\mathsf{Int}}(\langle x \mapsto [1,2]\rangle)) \\
&= \alpha_{\mathsf{Int}}([\![x - x]\!]\langle x \mapsto \{1,2\}\rangle) \\
&= \alpha_{\mathsf{Int}}(\langle x \mapsto 0\rangle) \\
&= \langle x \mapsto [0,0]\rangle
\end{aligned}$$

so that for a Boolean expression such as $(x - x > 0) \in \mathrm{BExp}$ trivially we get $[\![x - x > 0]\!]^{\mathsf{Int}}\langle x \mapsto [1,2]\rangle = \bot_{\mathsf{Int}}$. We can also define a sound compositional

$$
\begin{aligned}
[\![\mathbf{skip}]\!]^A S^\sharp &\triangleq S^\sharp \\
[\![x_i := a]\!]^A S^\sharp &\triangleq \alpha_A(\{s[x_i \mapsto (\!|a|\!)s] \mid s \in \gamma_A(S^\sharp)\}) \\
[\![C_1; C_2]\!]^A S^\sharp &\triangleq [\![C_2]\!]^A [\![C_1]\!]^A S^\sharp \\
[\![\mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2]\!]^A S^\sharp &\triangleq [\![C_1]\!]^A [\![b]\!]^A S^\sharp \sqcup_A [\![C_2]\!]^A [\![\neg b]\!]^A S^\sharp \\
[\![\mathbf{while}\ b\ \mathbf{do}\ C]\!]^A S^\sharp &\triangleq [\![\neg b]\!]^A \big(\mathsf{lfp}(\lambda X^\sharp.\, S^\sharp \sqcup_A [\![C]\!]^A [\![b]\!]^A X^\sharp)\big)
\end{aligned}
$$

Figure 3.5: Abstract denotational program semantics

abstract evaluation $[\![x - x]\!]^{\sharp_{\mathsf{Int}}}$ using the abstract sound operators $+_{\mathsf{Int}}$ and $*_{\mathsf{Int}}$ on Int which do not rely on the bca, such that we have the following sound compositional abstract evaluations:

$$[\![x - x]\!]^{\sharp_{\mathsf{Int}}}\langle x \mapsto [1,2]\rangle = [1,2] +_{\mathsf{Int}} ([1,2] *_{\mathsf{Int}} [-1,-1]) = [-1,1] \geq_{\mathsf{Int}} [0,0]$$

which leads to $[\![x - x > 0]\!]^{\sharp_{\mathsf{Int}}}\langle x \mapsto [1,2]\rangle = [1,1] \geq_{\mathsf{Int}} \bot_{\mathsf{Int}}$. ■

The compositional abstract semantics of programs $[\![C]\!]^A : A \to A$ is shown in Figure 3.5. It is defined by structural induction on programs and provides a correct approximation of the concrete collecting semantics. Recall that in a GI, the abstract join $\sqcup_A$ is always the best correct approximation of the concrete join $\cup$. Let us comment further on the definition of the abstract denotational program semantics in Figure 3.5:

(i) $[\![x_i := a]\!]^A$ is defined as best correct approximation of $\lambda S.\, [\![x_i := a]\!]S$ on A and does not rely on the best correct abstract semantics $[\![a]\!]^A$ of a. With reference to the Example 3.6, we have that $[\![x_2 := x_1 - x_1]\!]^{\mathsf{Int}}\langle x_1 \mapsto [1,2], x_2 \mapsto \top_{\mathsf{Int}}\rangle = \langle x_1 \mapsto [1,2], x_2 \mapsto [0,0]\rangle$, which is strictly more precise than $[\![x_2 := x_1 - x_1]\!]^{\sharp_{\mathsf{Int}}}\langle x_1 \mapsto [1,2], x_2 \mapsto \top_{\mathsf{Int}}\rangle = \langle x_1 \mapsto [1,2], x_2 \mapsto [-1,1]\rangle$.

(ii) $[\![C_1; C_2]\!]^A$ is compositionally defined and, in general, does not coincide with the best correct approximation of $\lambda S.\, [\![C_1; C_2]\!]S$, because the composition of best correct approximations, in general, does not provide the best correct approximation. On the other hand, it is worth remarking that this compositional definition preserves completeness, because, as recalled in Section 3.2, the composition of complete functions is complete.

(iii) $[\![\mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2]\!]^A$ is compositionally defined and relies on the lub $\sqcup_A$ of the abstract domain, which is the complete (therefore best correct) approximation of the concrete lub on $\wp^{\mathrm{re}}(\mathbb{S})$ (i.e., set union), because

the abstraction map of a GC is always additive. Thus, while this definition of $[\![\textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2]\!]^A$ is not the best correct approximation of the concrete version $\lambda S.[\![\textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2]\!]$, it preserves completeness, meaning that if $[\![b]\!]^A$, $[\![\neg b]\!]^A$, $[\![C_1]\!]^A$ and $[\![C_2]\!]^A$ are complete for their respective concrete counterparts then $[\![\textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2]\!]^A$ is a complete abstraction of the concrete of $\lambda S.[\![\textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2]\!]$.

(iv) The definition of $[\![\textbf{while } b \textbf{ do } C]\!]^A S^\sharp$ is based on the least fixpoint of the abstract function $\lambda X^\sharp.\, S^\sharp \sqcup_A [\![C]\!]^A [\![b]\!]^A X^\sharp : A \to A$, which is a compositional correct approximation on A of the concrete function $\lambda T.\, \gamma(S^\sharp) \cup [\![C]\!][\![b]\!]T$ used for defining $[\![\textbf{while } b \textbf{ do } C]\!]$. Similarly to point (iii), in general, this is not the best correct approximation, but it preserves completeness: thus, if $[\![b]\!]^A$ and $[\![C]\!]^A$ are complete then $\lambda X^\sharp.\, S^\sharp \sqcup_A [\![C]\!]^A [\![b]\!]^A X^\sharp$ is complete, so that by fixpoint transfer, completeness is transferred to the least fixpoint.

(v) It is easy to check (by structural induction on C) that the abstract semantics in Figure 3.5 is monotonic, namely, for all $C \in \mathsf{Prog}$, if $S_1^\sharp \leq_A S_2^\sharp$ then $[\![C]\!]^A S_1^\sharp \leq_A [\![C]\!]^A S_2^\sharp$.

It is well known that the abstract semantics in Figure 3.5 is correct, that is, for all $C \in \mathsf{Prog}$ and $S^\sharp \in A$,

$$[\![C]\!]\gamma_A(S^\sharp) \subseteq \gamma_A([\![C]\!]^A S^\sharp).$$

Example 3.7. Consider the following program Z:

$$\begin{array}{l} x := 9; \\ \textbf{while } x > 1 \textbf{ do} \\ \quad x := x - 1 \end{array}$$

Let us consider the interval abstract domain $\mathsf{Int} \in \mathscr{A}(\wp^{\mathrm{re}}(\mathbb{Z}))$. Let $S^\sharp \triangleq \langle x \mapsto [9,9]\rangle = \alpha_{\mathsf{Int}}(\{\langle x \mapsto 9\rangle\}) = [\![x := 9]\!]^{\mathsf{Int}}\langle x \mapsto \top_{\mathsf{Int}}\rangle$. Here, it turns out that the Kleene iterates of

$$\lambda X^\sharp.\, S^\sharp \sqcup_{\mathsf{Int}} [\![x := x - 1]\!]^{\mathsf{Int}} [\![x > 1]\!]^{\mathsf{Int}} X^\sharp$$

converge in a finite number of steps:

$$\begin{aligned} \langle x \mapsto \bot_{\mathsf{Int}}\rangle &\Rightarrow \langle x \mapsto [8,9]\rangle \\ &\Rightarrow \langle x \mapsto [7,9]\rangle \\ &\Rightarrow \dots \\ &\Rightarrow \langle x \mapsto [2,9]\rangle \\ &\Rightarrow \langle x \mapsto [1,9]\rangle. \end{aligned}$$

Therefore, the abstract semantics of program Z of every abstract input $S^\sharp \in \mathsf{Int}$ is $[\![Z]\!]^{\mathsf{Int}} S^\sharp = \langle x \mapsto [1,1]\rangle$ which corresponds to $\gamma_{\mathsf{Int}}(\langle x \mapsto [1,1]\rangle) = \{\langle x \mapsto 1\rangle\}$. Note that, in this case, for all $S \in \wp^{\mathrm{re}}(\mathbb{Z})$, we have

$$[\![Z]\!]S = \gamma_{\mathsf{Int}}([\![Z]\!]^{\mathsf{Int}}\alpha_{\mathsf{Int}}(S)) = \{\langle x \mapsto 1\rangle\}.$$

■

It is worth noticing that for a given loop **while** b **do** C, the finite convergence of the Kleene iterates of the concrete operator $\lambda T.\, \gamma(S^\sharp) \cup [\![C]\!][\![b]\!]T$ and of the abstract operator $\lambda X^\sharp.\, S^\sharp \sqcup_A [\![C]\!]^A [\![b]\!]^A X^\sharp$ are not logically related, as shown by the following example.

Example 3.8. Let us first observe that if A is some ACC abstract domain then we have that $\mathsf{lfp}(\lambda X^\sharp.\, S^\sharp \sqcup_A [\![C]\!]^A [\![b]\!]^A X^\sharp)$ always finitely converges, while the corresponding concrete fixpoint computation $\mathsf{lfp}(\lambda T.\, \gamma(S^\sharp) \cup [\![C]\!][\![b]\!]T)$ could not finitely converge (e.g., $[\![\textbf{while t do } x := x+1]\!]$ for some finite $\gamma(S^\sharp)$). It is more interesting to observe the other way round by considering the following program[1]:

$$\begin{array}{l} P_{\mathrm{col}} \triangleq\ x := 4; \\ \qquad \textbf{while } x > 1 \textbf{ do} \\ \qquad\quad \textbf{if } (x \bmod 2 = 0) \textbf{ then } x := x/2 \\ \qquad\quad \textbf{else } x := x * 3 + 1 \end{array}$$

where we assume to have the integer modulo operation mod and the division operator / where the results get rounded up in order to get an integer value. Let C denote the body of the while-loop of P_{col} and let $S \triangleq \{\langle x \mapsto 4\rangle\} = [\![x := 4]\!]\mathbb{Z}$. Here, we have that the Kleene iterates of $\lambda T. S \cup [\![C]\!][\![x > 1]\!]T$ in $\wp^{\mathrm{re}}(\mathbb{Z})$ converge in a finite number of steps:

$$\varnothing \Rightarrow \{\langle x \mapsto 4\rangle\} \Rightarrow \{\langle x \mapsto 4\rangle, \langle x \mapsto 2\rangle\} \Rightarrow \{\langle x \mapsto 4\rangle, \langle x \mapsto 2\rangle, \langle x \mapsto 1\rangle\}.$$

For the abstract semantics, let us consider the interval abstract domain Int and let $S^\sharp \triangleq \langle x \mapsto [4,4]\rangle = [\![x := 4]\!]^{\mathsf{Int}}\langle x \mapsto \top_{\mathsf{Int}}\rangle$. Here, it turns out that the Kleene iterates of $\lambda X^\sharp.\, S^\sharp \sqcup_{\mathsf{Int}} [\![C]\!]^{\mathsf{Int}}[\![x > 1]\!]^{\mathsf{Int}} X^\sharp$ do not finitely converge:

$$\begin{aligned} \langle x \mapsto \bot_{\mathsf{Int}}\rangle &\Rightarrow \langle x \mapsto [2,4]\rangle \\ &\Rightarrow \langle x \mapsto [1,10]\rangle \\ &\Rightarrow \langle x \mapsto [1,19]\rangle \\ &\Rightarrow \langle x \mapsto [1,58]\rangle \\ &\Rightarrow \cdots \end{aligned}$$

[1] The corresponding function (without the statement $x := 4$) is also known in the literature as the Collatz function which forms the *Collatz conjecture*, also known as the $3n+1$ problem. For a comprehensive coverage, the reader is referred to [54]

Let us explain the step $\langle x \mapsto [1,10]\rangle \Rightarrow \langle x \mapsto [1,19]\rangle$, which, by assuming the best correct approximations of Boolean expressions on intervals, is a consequence of the following abstract evaluations:

$$\begin{aligned}
&[\![x > 1]\!]^{\mathsf{Int}}\langle x \mapsto [1,10]\rangle = \langle x \mapsto [2,10]\rangle\\
&[\![x \bmod 2 = 0]\!]^{\mathsf{Int}}\langle x \mapsto [2,10]\rangle = \langle x \mapsto [2,10]\rangle\\
&\qquad [\![x := x/2]\!]^{\mathsf{Int}}\langle x \mapsto [2,10]\rangle = \langle x \mapsto [1,5]\rangle\\
&[\![\neg(x \bmod 2 = 0)]\!]^{\mathsf{Int}}\langle x \mapsto [2,10]\rangle = \langle x \mapsto [3,9]\rangle\\
&\qquad [\![x := x * 3 + 1]\!]^{\mathsf{Int}}\langle x \mapsto [3,9]\rangle = \langle x \mapsto [10,19]\rangle\\
&\langle x \mapsto [1,5]\rangle \sqcup_{\mathsf{Int}} \langle x \mapsto [10,19]\rangle = \langle x \mapsto [1,19]\rangle
\end{aligned}$$

Hence, on an abstract domain A which is not ACC, such as Int, it may well happen that the concrete Kleene iterates finitely converge while the corresponding abstract Kleene iterates on A do not converge in a finite number of steps. ■

By observing the Example 3.8, we can conclude that the abstract semantics defined in Figure 3.5, in general, does not define a static program analysis. Indeed, static program analyzers must always terminate on all inputs $\alpha_A(S)$ if the concrete collecting semantics terminates on it (i.e. $[\![P]\!]S \neq \varnothing$) and therefore they correctly approximate the least fixpoint of the abstract function $\lambda X^\sharp.\, S^\sharp \sqcup_A [\![C]\!]^A [\![b]\!]^A X^\sharp$. This can be achieved by replacing, at some iteration, the abstract lub $\sqcup_A$ of its Kleene iterates with a widening operator ∇. A binary operator $\nabla : A \times A \to A$ is a widening operator [26] in an abstract domain of stores $A \in \mathscr{A}(\wp^{\mathrm{re}}(\mathbb{S}))$ if: (i) it computes upper bounds, namely $\forall a, b \in A.\ a \leq_A a \nabla b$ and $b \leq_A a \nabla b$; (ii) and for all sequences $(b^i)_{i\in\mathbb{N}}$ in A, the sequence $(a^i)_{i\in\mathbb{N}}$ computed as $a^0 \triangleq b^0$, $a^{i+1} \triangleq a^i \nabla b^{i+1}$ stabilizes after a finite number of steps, namely, there exists $k \geq 0$ such that $a^{k+1} = a^k$. Widenings can indeed be used to approximate least fixpoints in abstract domains. If f is a monotonic operator in a complete concrete lattice and $f^\sharp$ is a sound abstraction of f, then the following iteration

$$\begin{aligned}
a^0 &\triangleq \bot_A\\
a^{i+1} &\triangleq a^i \nabla f^\sharp(a^i)
\end{aligned}$$

converges in finitely many steps, and its limit a^{fix} is a sound abstraction of the least fixpoint $\mathsf{lfp}(f) : \mathsf{lfp}(f) \leq_C \gamma_A(a^{\mathrm{fix}})$. Therefore, a widening operator is used to accelerate or force the convergence of Kleene iterates in the abstract domain A, at the cost of losing the property of having the best correct abstract semantics of programs. From now on, we will assume the following

Assumption 3. *Given an abstract domain of stores $A \in \mathscr{A}(\wp^{\mathrm{re}}(\mathbb{S}))$, we only consider static program analysis $[\![C]\!]^A : A \to A$ that terminates in a finite number of steps on all inputs $S^\sharp \in A$.*

It is worth to note that the soundness of abstract interpretation and Assumption 3 imply that for every program $P \in \mathsf{Prog}$ and set of stores $S \in \wp^{\mathrm{re}}(\mathbb{S})$:

$$[\![P]\!]S \neq \varnothing \;\Rightarrow\; [\![P]\!]^A \alpha_A(S) \neq \bot_A.$$

This assumption is essential in order for the abstract interpreter to soundly approximate all possible concrete executions. Note that this condition is also trivially satisfied by abstract domains of stores with a finite number of elements or that meet the ACC.

It is worth remarking that, the abstract semantics defined in Figure 3.5 is a correct approximation of the concrete collecting semantics defined in Figure 2.2, *but*, in general, not the bca which is defined as: $\gamma_A(\alpha_A([\![P]\!]\gamma_A(\alpha_A(S))))$. Moreover, for all $A \in \mathscr{A}(\wp^{\mathrm{re}}(\mathbb{S}))$, $P \in \mathsf{Prog}$ and $S \in \wp^{\mathrm{re}}(\mathbb{S})$, the following inequalities hold:

$$\alpha_A([\![P]\!]S) \;\leq_A\; \alpha_A([\![P]\!]\gamma_A(\alpha_A(S))) \;\leq_A\; [\![P]\!]^A \alpha_A(S).$$

3.5 Recursive Abstract Domains of Stores

Static program analysis always relies upon recursive, namely decidable, abstractions. This is because the analysis of programs requires decidable answers to undecidable questions such as those expressed as extensional properties of programs and concerning their dynamic behaviour. The notion of decidable or recursive abstract interpretation has been studied in [30] for the comparison of the difficulty in analyzing and verifying programs. In the following, we formalize this notion for a generic GC based abstract interpretation.

Definition 3.9 (Recursive abstract domains of stores). *An abstract domain of stores $A \in \mathscr{A}(\wp^{\mathrm{re}}(\mathbb{S}))$ is recursive if:*

(i) for each store $s \in \mathbb{S}$, $\alpha_A(\{s\})$ is computable;

(ii) for all $S^\sharp \in A$, the set of stores $\gamma_A(S^\sharp)$ is recursive, namely, $\gamma_A(S^\sharp) \in \wp^{rec}(\mathbb{S})$;

(iii) the partial order relation $\leq_A$ is decidable, i.e., there exists an always defined program (total recursive function) for deciding if $a_1 \leq_A a_2$ holds, for all $a_1, a_2 \in A$.

Note that (iii) implies that the equivalence relation between abstract elements (i.e., recursive sets of stores) is decidable. Indeed, by the antisymmetry property satisfied by all partial order relations, if $a_1 \leq_A a_2$ and $a_2 \leq_A a_1$ then $a_1 = a_2$. The intuition is that the elements of any recursive abstract domain of stores represent recursive sets of stores, therefore, they are isomorphic to a suitable

subset of $\wp^{\mathrm{rec}}(\mathbb{S})$. Besides recursive abstract domains, we will also consider trivial abstractions of stores.

Definition 3.10 (Trivial abstractions of stores). *The trivial abstractions of stores in $\mathscr{A}(\wp^{\mathrm{re}}(\mathbb{S}))$ are:*

- *$id \in \mathscr{A}(\wp^{\mathrm{re}}(\mathbb{S}))$ for the (least) identical abstraction of stores such that for all $S \in \wp^{\mathrm{re}}(\mathbb{S})$, $\alpha_{id}(S) = S = \gamma_{id}(S)$, and*
- *$\top^{\mathbb{S}} \in \mathscr{A}(\wp^{\mathrm{re}}(\mathbb{S}))$ for the the greatest abstraction of stores such that for all $S \in \wp^{\mathrm{re}}(\mathbb{S})$, $\alpha_{\top^{\mathbb{S}}}(S) = \mathbb{S}$.*

Thus, an abstract domain of stores $A \in \mathscr{A}(\wp^{\mathrm{re}}(\mathbb{S}))$ is defined to be trivial if $A \in \{id, \top^{\mathbb{S}}\}$.

In the following we will only consider program analyzers as specified by Galois insertion-based abstract interpreters (Assumption 2) over strict recursive or trivial abstract domains of stores. This assumption is often left implicit in program analysis. We emphasize this hypothesis through the following

Assumption 4. *In the rest of the thesis, the abstract domains of stores considered in $\mathscr{A}(\wp^{\mathrm{re}}(\mathbb{S}))$ are assumed either strict recursive or trivial.*

As a consequence of Assumption 4, any non-trivial abstract domain is assumed recursive and strict. Note that $\top^{\mathbb{S}}$ is a recursive abstract domain but not strict, while id is strict but not recursive because $\gamma_{id}(S)$ might not be recursive, thus conditions (ii) and (iii) of Definition 3.9 are not satisfied. The GI-based abstract interpretations and the strictness conditions allow us to consider non-trivial abstract domains of stores that exactly represent the empty set $\varnothing$.

Proposition 3.11. *For every non-trivial $A \in \mathscr{A}(\wp^{\mathrm{re}}(\mathbb{S}))$, $\gamma_A(\alpha_A(\varnothing)) = \varnothing$.*

Proof. By Assumption 2, we know that (1) $\alpha_A \circ \gamma_A = id$ and, by Assumption 4, (2) $\gamma_A(\bot_A) = \varnothing$. Therefore:

$$\begin{aligned}
\gamma_A(\bot_A) = \varnothing &\Leftrightarrow \alpha_A(\gamma_A(\bot_A)) = \alpha_A(\varnothing) \\
&\Leftrightarrow \bot_A = \alpha_A(\varnothing) && \text{[by (1)]} \\
&\Leftrightarrow \gamma_A(\bot_A) = \gamma_A(\alpha_A(\varnothing)) \\
&\Leftrightarrow \varnothing = \gamma_A(\alpha_A(\varnothing)) && \text{[by (2)]}
\end{aligned}$$

□

3.6 Completeness Class of Programs

A program $P \in \mathsf{Prog}$ is complete for the store abstraction $A \in \mathscr{A}(\wp^{\mathrm{re}}(\mathbb{S}))$ when for all $S \in \wp^{\mathrm{re}}(\mathbb{S})$:

$$\alpha_A([\![P]\!]S) = [\![P]\!]^A \alpha_A(S).$$

The set of all complete programs for a given abstraction of stores, forms the completeness class and can be represented as a function from abstractions to programs, namely $\mathbb{C} : \mathscr{A}(\wp^{\mathrm{re}}(\mathbb{S})) \to \wp(\mathsf{Prog})$.

Definition 3.12 (Completeness class). *Given $A \in \mathscr{A}(\wp^{\mathrm{re}}(\mathbb{S}))$, its completeness class $\mathbb{C}(A) \subseteq \mathsf{Prog}$ is defined as:*

$$\mathbb{C}(A) \triangleq \{P \in \mathsf{Prog} \mid \forall S \in \wp^{\mathrm{re}}(\mathbb{S}).\ \alpha_A([\![P]\!]S) = [\![P]\!]^A \alpha_A(S)\}.$$

The set $\mathbb{C}(A)$ is an infinite and non-extensional program property. In fact, $\mathbb{C}(A)$ is infinite by a straightforward padding argument, since **skip** $\in \mathbb{C}(A)$ for every A and sequential composition of complete commands is still complete. Also, $\mathbb{C}(A)$ is non-extensional because there always exist programs P and Q such that: P is complete for A, $[\![P]\!] = [\![Q]\!]$, and Q is not complete for A. This phenomenon is known in static analysis where semantics preserving program transformations may lose/increase precision of analyses (e.g., see [42, 55]).

Example 3.13. Consider the programs R and Z presented respectively in Examples 2.1 and 3.7. It is easy to observe that programs R and Z are extensionally equivalent, i.e., for every $S \in \wp^{\mathrm{re}}(\mathbb{Z})$, $[\![Z]\!]S = [\![R]\!]S = \{\langle x \mapsto 1\rangle\}$. However, interpreted in Int, R and Z exhibit different abstract semantics. We have seen in Example 3.7 that the integer interval domain Int is complete for abstract interpretation of the program Z because $[\![Z]\!]S = \gamma_A([\![Z]\!]^{\mathsf{Int}}\alpha_{\mathsf{Int}}(S)) = \{\langle x \mapsto 1\rangle\}$. Whereas, it is easy to see that, interpreting R in Int, we get the following Kleene iterates of $\lambda X^\sharp.\ S^\sharp \sqcup_{\mathsf{Int}} [\![x := x - 2]\!]^{\mathsf{Int}}[\![x > 1]\!]^{\mathsf{Int}}X^\sharp$:

$$\begin{aligned}\langle x \mapsto \bot_{\mathsf{Int}}\rangle &\Rightarrow \langle x \mapsto [7,9]\rangle\\ &\Rightarrow \langle x \mapsto [5,9]\rangle\\ &\Rightarrow \langle x \mapsto [3,9]\rangle\\ &\Rightarrow \langle x \mapsto [1,9]\rangle\\ &\Rightarrow \langle x \mapsto [0,9]\rangle.\end{aligned}$$

Therefore, for all $S \in \wp^{\mathrm{re}}(\mathbb{Z})$, we have

$$\gamma_{\mathsf{Int}}([\![R]\!]^{\mathsf{Int}}\alpha_{\mathsf{Int}}(S)) = \gamma_{\mathsf{Int}}(\langle x \mapsto [0,1]\rangle) \supset \{\langle x \mapsto 1\rangle\} = [\![R]\!]S.$$

This shows that the integer interval domain Int is incomplete for abstract interpretation of the program R, namely $R \notin \mathbb{C}(\mathsf{Int})$, while it remains complete for program Z, namely $Z \in \mathbb{C}(\mathsf{Int})$. ■

It turns out that the relative precision of abstract domains—encoded by the ordering $\leq_{\mathscr{A}(\wp^{\mathrm{re}}(\mathbb{S}))}$ on the lattice of abstractions—and the corresponding classes of completeness are not related. In particular, it may well happen that for an abstraction of stores $A \in \mathscr{A}(\wp^{\mathrm{re}}(\mathbb{S}))$ which is complete for a given program $P \in \mathsf{Prog}$, a refinement $A^{\mathrm{ref}} \in \mathscr{A}(\wp^{\mathrm{re}}(\mathbb{S}))$ of A, i.e., $A^{\mathrm{ref}} \leq_{\mathscr{A}(\wp^{\mathrm{re}}(\mathbb{S}))} A$, turns out to be instead incomplete for the same program P. This phenomenon is well known in static program analysis and it corresponds to the fact that coarse abstractions may induce a complete static analysis for some program where more precise ones instead fail to be complete for that same program.

Chapter 4

PARTIAL COMPLETENESS

In this section, we formalize the notion of *partial completeness*, a weaker concept of the known completeness in field of abstract interpretation. The notion of partial completeness is useful for *limiting* imprecision in abstract interpretation. More specifically, we present a formal framework that makes use of *quasi-metrics* defining a distance model, to compare the elements of the abstract domains of stores according to their precision (Section 4.1). We formalize ε-partial completeness of an abstract domain of stores A using a quasi-metric A-compatible, to describe the ε-bounded level of imprecision. On this basis, in Section 4.2 we introduce the notion of ε-partial completeness class w.r.t. an abstract quasi-metric space of stores, to enclose the set of all programs for which the abstract interpreter never outputs an abstract value whose quasi-metric distance from the concrete ones exceeds ε. We then focus on the investigation of the computational limits of the class of partially complete programs with respect to a given abstract quasi-metric space of stores (Section 4.3). We conclude by formalizing in Section 4.4 a less restrictive completeness class, called local ε-partial completeness, which requires ε-partial completeness on a fixed set of inputs only, making it more "tractable" in terms of computability w.r.t. ε-partial completeness class when A satisfies the ACC condition. This chapter is mainly based on [10].

4.1 Quasi-metrics on Abstract Domains

Our goal is to introduce a new perspective that allows us to bound the imprecision of an abstract interpreter with respect to a given measure between abstract outputs. Therefore, first of all, we need a metric in order to compare the elements of the abstract domain according to their precision. Roughly, a metric allows us

to measure the distance between elements of a given set. When applied to an abstract domain of stores $A \in \mathscr{A}(\wp^{\mathrm{re}}(\mathbb{S}))$, i.e., a set whose elements are related by a partial order, we would like to have a distance $d : A \times A \to \mathbb{R}_{\geq 0}$ which is somehow compatible with the underlying order $\leq_A$. For instance, if $a_1 \leq_A a_2 \leq_A a_3$ with $a_1, a_2, a_3 \in A$, then one would expects that $d(a_1, a_2) \leq d(a_1, a_3)$. The classical definition of distance in measure theory does not work in this situation. This is because, intuitively, in a metric space one wants to compare any two elements, while in a poset some of the objects are indeed incomparable. In this section, we aim at relaxing the classical notion of distance on sets and to conjugate it with the underlying order of the elements of an abstract domain of stores.

We weaken the notion of metric with the notion of *quasi-metric* [74]. A quasi-metric on a generic non-empty set S is a metric function $\delta : S \times S \to \mathbb{R}_{\geq 0}$ whose symmetry property may not hold. A set endowed with a quasi-metric is called *quasi-metric space*. In the following definition, we adapt the above standard definition of quasi-metric in order to cope with the structure of any abstract domain of stores $A \in \mathscr{A}(\wp^{\mathrm{re}}(\mathbb{S}))$.

Definition 4.1 (Quasi-metrics A-compatible). *Let $A \in \mathscr{A}(\wp^{\mathrm{re}}(\mathbb{S}))$ be an abstract domain of stores ordered according to the relation $\leq_A$. We say that $\delta_A : A \times A \to \mathbb{Q}_{\geq 0} \cup \{\bot, \infty\}$ is a quasi-metric A-compatible if for all $a_1, a_2, a_3 \in A$ and $\varepsilon \in \mathbb{Q}_{\geq 0}$, it satisfies the following axioms:*

(i) $\delta_A(a_1, a_2) \leq \varepsilon$ *is a decidable predicate*

(ii) $a_1 = a_2 \;\Leftrightarrow\; \delta_A(a_1, a_2) = 0$ *(identity of indiscernibles)*

(iii) $a_1 \leq_A a_2 \;\Leftrightarrow\; \delta_A(a_1, a_2) \neq \bot$ *(approximation)*

(iv) $a_1 \leq_A a_2 \leq_A a_3 \;\Rightarrow\; \delta_A(a_1, a_3) \leq \delta_A(a_1, a_2) + \delta_A(a_2, a_3)$ *(weak triangle inequality)*

Here, for all $\varepsilon \in \mathbb{Q}_{\geq 0}$: $\varepsilon < \infty$. We allow the quasi-metric between two elements to be $\bot$, which represents an undefined distance. It is worth noting that, by considering only recursive or trivial abstract domains of stores (Assumption 4), any abstract domain of stores $A \in \mathscr{A}(\wp^{\mathrm{re}}(\mathbb{S}))$ contains a finite or countably infinite number of r.e. sets. This means that $|A| \leq \omega = |\mathbb{N}| = |\mathbb{Q}_{\geq 0}|$. Therefore, it is reasonable to consider the range of a quasi-metric A-compatible function over $\mathbb{Q}_{\geq 0}$ instead of the positive real numbers as in the original definition of quasi-metric. Informally, we say that a_1 is approximated by a_2 if and only if the quasi-metric A-compatible between a_1 and a_2 is not $\bot$. Therefore, the value of the distance δ_A between two elements can be interpreted as the error introduced by the approximation: *the lower is the value of the error $\delta_A(a_1, a_2)$, the better is the approximation*. The value $\delta_A(a_1, a_2) = \bot$ expresses naturally the fact that a_2 does not approximate a_1, i.e., $a_1 \not\leq_A a_2$, while equal elements will always have a null quasi-distance. Note that, for every quasi-metric A-compatible, the

approximation axiom implies δ_A to embody the underlying poset structure and therefore $\leq_A$ induces the quasi-metric δ_A. We can now adapt the general definition of quasi-metric space to the definition of *abstract quasi-metric space of stores.*

Definition 4.2 (Abstract quasi-metric spaces of stores). *An abstract domain of stores $A \in \mathscr{A}(\wp^{\mathrm{re}}(\mathbb{S}))$ endowed with a quasi-metric A-compatible δ_A, forms an abstract quasi-metric space, denoted by $\boldsymbol{A} \triangleq (A, \delta_A)$. We use $\mathfrak{A}(\wp^{\mathrm{re}}(\mathbb{S}))$ to refer to the the set of all abstract quasi-metric spaces of stores.*

As an example, we formalize the *weighted path-length* distance which considers the lattice A as a weighted directed graph. Let $E_A \subseteq A \times A$ be the set of all pairs (a, b) such that $a \lessdot_A b$. Let $a, b \in A$ with $a \neq b$, we denote with $\mathfrak{C}_a^b$ the set of all possible chains $\mathbf{c} \subseteq E$ such that if $(c, d) \in \mathbf{c}$ then $a \leq_A c \lessdot_A d \leq_A b$. It is clear that if $a \not\leq_A b$ then $\mathfrak{C}_a^b = \varnothing$.

Definition 4.3 (Weighted path-length). *Let $\mathfrak{w} : E_A \to \mathbb{R}_{>0}$ be a weight function. We define $\delta^{\mathfrak{w}} : A \times A \to \mathbb{Q}_{\geq 0} \cup \{\infty, \bot\}$ with $A \in \mathscr{A}(\wp^{\mathrm{re}}(\mathbb{S}))$ such that for every $a, b \in A$:*

$$\delta^{\mathfrak{w}}(a,b) \triangleq \begin{cases} 0 & \text{if } a = b, \\ \infty & \text{if } \forall \mathbf{c} \in \mathfrak{C}_a^b.\ |\mathbf{c}| = \omega, \\ min\left\{ \sum_{e \in \mathbf{c}} \mathfrak{w}(e) \;\middle|\; \begin{array}{l} \mathbf{c} \in \mathfrak{C}_a^b \\ |\mathbf{c}| < \omega \end{array} \right\} & \text{if } \exists \mathbf{c} \in \mathfrak{C}_a^b.\ |\mathbf{c}| < \omega, \\ \bot & \text{if } \mathfrak{C}_a^b = \varnothing. \end{cases}$$

□

It is easy to note that the previous definition fulfills axioms (i)-(iv) of Definition 4.1.

Proposition 4.4. *For any $A \in \mathscr{A}(\wp^{\mathrm{re}}(\mathbb{S}))$, $\delta_A^{\mathfrak{w}}$ is a quasi-metric A-compatible.*

Intuitively, $\delta_A^{\mathfrak{w}}$ reflects exactly the underlying abstract domain structure. It counts the minimum weighted path w.r.t. $\mathfrak{w}$ of intermediate elements between two comparable elements of an abstract domain A. Note that $\delta_A^{\mathfrak{w}}$ does not satisfy symmetry and it relates only elements that belong to the same chain. It is clear that $\delta_A^{\mathfrak{w}}$ refines the standard metric associated with the partial order on A, providing a quantitative value to the length of chains separating abstract objects in A. The following are examples of $\delta_A^{\mathfrak{w}}$ instantiated for $A \in \{\mathsf{Sign}, \mathsf{P} \sqcap \mathsf{S}, \mathsf{Int}\}$.

Example 4.5. Consider the $\mathsf{Sign} \triangleq \{\mathbb{Z}, -, 0, +, \varnothing\}$ abstract domain of stores shown in Example 3.1. For any $(a, b) \in E_{\mathsf{Sign}}$, let $\mathfrak{w}(a, b) \triangleq 1$. Then, the weighted path-length is a quasi-metric Sign-compatible and the couple $(\mathsf{Sign}, \delta_{\mathsf{Sign}}^{\mathfrak{w}}) \in \mathfrak{A}(\wp^{\mathrm{re}}(\mathbb{Z}))$

forms an abstract quasi-metric space of stores ranging over one variable. The following are examples of evaluations of $\delta^{\mathfrak{w}}_{\mathsf{Sign}}$ on some elements of Sign:

$$\delta^{\mathfrak{w}}_{\mathsf{Sign}}(+,+) = 0 \qquad \delta^{\mathfrak{w}}_{\mathsf{Sign}}(\varnothing,0) = 1 \qquad \delta^{\mathfrak{w}}_{\mathsf{Sign}}(0,\mathbb{Z}) = 2$$
$$\delta^{\mathfrak{w}}_{\mathsf{Sign}}(\varnothing,\mathbb{Z}) = 3 \qquad \delta^{\mathfrak{w}}_{\mathsf{Sign}}(\mathbb{Z},\varnothing) = \bot \qquad \delta^{\mathfrak{w}}_{\mathsf{Sign}}(-,+) = \bot.$$

Note that $\delta^{\mathfrak{w}}_{\mathsf{Sign}}(\varnothing,\mathbb{Z}) \neq \delta^{\mathfrak{w}}_{\mathsf{Sign}}(\mathbb{Z},\varnothing)$, hence, $\delta^{\mathfrak{w}}_{\mathsf{Sign}}$ does not satisfy the symmetry axiom. ■

Example 4.6. Consider the $\mathsf{Parity} \triangleq \{\mathbb{Z}, \mathit{even}, \mathit{odd}, \varnothing\}$ abstract domain shown in Figure 3.1. For any $(a,b) \in E_{\mathsf{Parity}}$, let $\mathfrak{w}(a,b) \triangleq 1$. Then, the weighted path-length is a quasi-metric Parity-compatible and the couple $(\mathsf{Parity}, \delta^{\mathfrak{w}}_{\mathsf{Parity}}) \in \mathfrak{A}(\wp^{\mathrm{re}}(\mathbb{Z}))$ forms an abstract quasi-metric space of stores ranging over one variable. ■

Example 4.7. Consider the abstract domain $\mathsf{P} \sqcap \mathsf{S} \in \mathscr{A}(\wp(\mathbb{Z}))$, shown in Fig. 1.3. $\mathsf{P} \sqcap \mathsf{S}$ is a recursive non-relational store abstraction in $\mathscr{A}(\wp^{\mathrm{re}}(\mathbb{Z}))$ [27], that satisfies both the ACC and DCC properties. Let $\mathfrak{w}(\varnothing,(\mathit{odd},+)) = \mathfrak{w}(\varnothing,(\mathit{odd},-)) \triangleq 2$ while 1 for the remaining pairs in $E_{\mathsf{P}\sqcap\mathsf{S}}$. Then, the weighted path-length $\delta^{\mathfrak{w}}_{\mathsf{P}\sqcap\mathsf{S}}$ is a quasi-metric $\mathsf{P} \sqcap \mathsf{S}$-compatible and the pair $(\mathsf{P} \sqcap \mathsf{S}, \delta^{\mathfrak{w}}_{\mathsf{P}\sqcap\mathsf{S}}) \in \mathfrak{A}(\wp^{\mathrm{re}}(\mathbb{Z}))$ forms an abstract quasi-metric space of stores ranging over one variable. ■

Example 4.8. Consider the abstract domain of intervals $\mathsf{Int} \in \mathscr{A}(\wp(\mathbb{Z}))$, shown in Example 3.3. For any $(a,b) \in E_{\mathsf{Int}}$, let $\mathfrak{w}(a,b) \triangleq 1$. Then, the weighted path-length $\delta^{\mathfrak{w}}_{\mathsf{Int}}$ is a quasi-metric Int-compatible and the pair $(\mathsf{Int}, \delta^{\mathfrak{w}}_{\mathsf{Int}}) \in \mathfrak{A}(\wp^{\mathrm{re}}(\mathbb{Z}))$ forms an abstract quasi-metric space of stores ranging over one variable. The intuition of $\delta^{\mathfrak{w}}_{\mathsf{Int}}$ is to count how many more elements one interval has w.r.t. another one. That is, if $i_1, i_2 \in \mathsf{Int}$ and $\delta^{\mathfrak{w}}_{\mathsf{Int}}(i_1,i_2) = k$ for some $k \in \mathbb{N}$, then the interval i_2 contains exactly k more elements than i_1. ■

Let us fix some notations. From now on, we will use the bold symbol $\boldsymbol{A}$ indicating a pair of abstract domain of stores $A \in \mathscr{A}(\wp^{\mathrm{re}}(\mathbb{S}))$ together with a quasi-metric A-compatible $\delta_A : A \times A \to \mathbb{Q}_{\geq 0} \cup \{\bot, \infty\}$, whereas, when we want to use a specific quasi-metric A-compatible and / or a specific abstract domain A in (A, δ_A), we will write explicitly the pair, e.g., $(\mathsf{Sign}, \delta^{\mathfrak{w}}_{\mathsf{Sign}})$. If δ_A has no superscript, then it is intended as for every quasi-metric A-compatible, otherwise the superscript refers to the specific quasi-metric used. For example, δ_{Sign} refers to any quasi-metric Sign-compatible, whereas $\delta^{\mathfrak{w}}_{\mathsf{Sign}}$ corresponds to the weighted path-length quasi-metric Sign-compatible defined in Definition 4.3. In addition, when we say that $\boldsymbol{A}$ is trivial, non-trivial, ACC or DCC, we always refer to its abstract domain of stores A defining the pair $\boldsymbol{A} = (A, \delta_A)$.

4.2 Partially Complete Abstractions of Stores

Standard completeness, e.g., see [26, 27, 43], means that no false positives are returned by analyzing the program with an abstract interpreter. We introduce the notion of *partial completeness* in abstract interpretation. An ε-partially complete abstract interpretation *allows* some false alarms to be reported, *but* their number is bounded by a constant $\varepsilon \in \mathbb{Q}_{\geq 0}$. This constant is determined according to a quasi-metric which is A-compatible δ_A.

The notion of ε-*closeness* between abstract states, according to a A-compatible quasi-metric δ_A, specifies the boundaries of allowed imprecision in the abstract domain.

Definition 4.9 (ε-Closeness w.r.t. δ_A). *Let $\boldsymbol{A} \in \mathfrak{A}(\wp^{\mathrm{re}}(\mathbb{S}))$ be an abstract quasi-metric space of stores and consider a constant $\varepsilon \in \mathbb{Q}_{\geq 0}$. For all $a, b \in A$ such that $a \leq_A b$, we say that b is ε-close to a w.r.t. δ_A, denoted $a \approx^{\varepsilon}_{\delta_A} b$, if $\delta_A(a, b) \neq \bot$ and $\delta_A(a, b) \leq \varepsilon$.*

The concept of closeness encapsulates both the approximation and the relative error, guided by the quasi-metric δ_A. Therefore, two abstract states that are ε-close w.r.t. δ_A differ just for a maximum error quantified as ε.

Definition 4.10 (ε-Partial completeness). *Let $P \in \mathsf{Prog}$ be a program and $\varepsilon \in \mathbb{Q}_{\geq 0}$ a constant. An abstract quasi-metric space of stores $\boldsymbol{A} \in \mathfrak{A}(\wp^{\mathrm{re}}(\mathbb{S}))$ is ε-partially complete for P if for all $S \in \wp^{\mathrm{re}}(\mathbb{S})$:*

$$\alpha_A([\![P]\!]S) \approx^{\varepsilon}_{\delta_A} [\![P]\!]^A \alpha_A(S).$$

The general idea of partial completeness is depicted in Figure 4.1. We have all the ingredients to introduce the notion of ε-partial completeness class as a mapping from abstract quasi-metric spaces of stores $\boldsymbol{A}$ and a constant $\varepsilon \in \mathbb{Q}_{\geq 0}$, to the set of all programs for which the abstraction is ε-partially complete, i.e.,

$$\mathbb{C} : \mathfrak{A}(\wp^{\mathrm{re}}(\mathbb{S})) \times \mathbb{Q}_{\geq 0} \to \wp(\mathsf{Prog}).$$

Definition 4.11 (ε-Partial completeness class). *The partial completeness class of an abstract quasi-metric space of stores $\boldsymbol{A} \in \mathfrak{A}(\wp^{\mathrm{re}}(\mathbb{S}))$ and a constant $\varepsilon \in \mathbb{Q}_{\geq 0}$, denoted $\mathbb{C}(\boldsymbol{A}, \varepsilon) \subseteq \mathsf{Prog}$, is defined as:*

$$\mathbb{C}(\boldsymbol{A}, \varepsilon) \triangleq \{P \in \mathsf{Prog} \mid \forall S \in \wp^{\mathrm{re}}(\mathbb{S}).\ \alpha_A([\![P]\!]S) \approx^{\varepsilon}_{\delta_A} [\![P]\!]^A \alpha_A(S)\}.$$

Because recursive and trivial abstract domains of stores are either finite or countably infinite, namely, $\forall A \in \mathscr{A}(\wp^{\mathrm{re}}(\mathbb{S}))$. $|A| \leq \omega$, we can define $\mathbb{C}(\boldsymbol{A}, \infty) \triangleq \cup_{\varepsilon \in \mathbb{Q}_{\geq 0}} \mathbb{C}(\boldsymbol{A}, \varepsilon)$. If $P \in \mathbb{C}(\boldsymbol{A}, \varepsilon)$ then, for every input stores, all the possible collecting semantics and abstract semantics produced respectively by the concrete

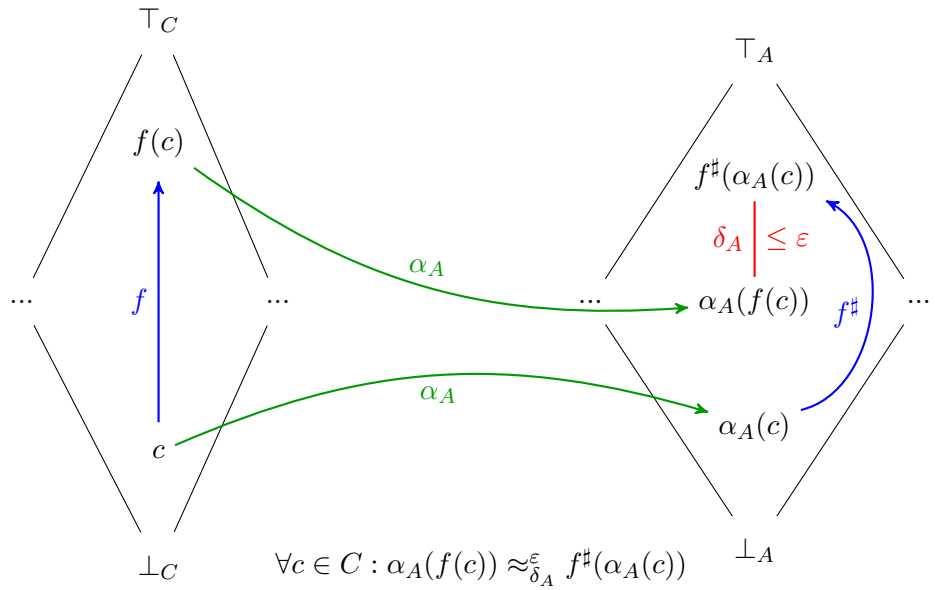


Figure 4.1: The general idea of partial completeness

interpreter and the abstract interpreter of P, are ε-close w.r.t. δ_A in the abstract domain. Roughly, a partial completeness class $\mathbb{C}(\boldsymbol{A}, \varepsilon)$ is defined to be the set of all programs whose abstract interpretation on a given store abstraction A *can* produce false alarms, *but* the error introduced is not greater then ε according to the A-compatible quasi-metric δ_A. The following Proposition is a straightforward result from Definitions 3.12 and 4.11 of completeness and partial completeness classes, and Definition 4.1 of A-compatible quasi-metric.

Proposition 4.12. *For all* $\boldsymbol{A} \in \mathfrak{A}(\wp^{\mathrm{re}}(\mathbb{S}))$*, the following hold:*

(i) $\mathbb{C}(\boldsymbol{A}, 0) = \mathbb{C}(A)$*;*

(ii) $\forall \varepsilon, \xi \in \mathbb{Q}_{\geq 0}.\ \varepsilon \leq \xi \ \Rightarrow\ \mathbb{C}(\boldsymbol{A}, \varepsilon) \subseteq \mathbb{C}(\boldsymbol{A}, \xi)$*;*

(iii) $\mathbb{C}(\boldsymbol{A}, \infty) = \mathsf{Prog}$.

Forcing a zero closeness means requiring no false alarms, i.e., standard completeness, while each complete abstraction is also partially complete. Moreover, weakening closeness increases monotonically the set of partially complete programs.

Example 4.13. Consider the abstract domain for sign analysis $\mathsf{Sign} \triangleq \{\mathbb{Z}, -, 0, +, \varnothing\}$. Let us consider the weighted path-length quasi-metric Sign-compatible $\delta^{\mathfrak{w}}_{\mathsf{Sign}}$ and the partial completeness class $\mathbb{C}((\mathsf{Sign}, \delta^{\mathfrak{w}}_{\mathsf{Sign}}), 1)$. Consider the following program P:

$$x := 1;$$
$$x := x + 1;$$
$$x := x - 1$$

such that $[\![P]\!] = \lambda S \in \wp^{\mathrm{re}}(\mathbb{Z}).\{\langle x \mapsto 1\rangle\}$ and $\alpha_{\mathsf{Sign}}([\![P]\!]S) = \lambda S \in \wp^{\mathrm{re}}(\mathbb{Z}).\langle x \mapsto +\rangle$. Let us observe that

$$[\![x := 1]\!]^{\mathsf{Sign}}\langle x \mapsto \mathbb{Z}\rangle = \langle x \mapsto +\rangle,$$
$$[\![x := x + 1]\!]^{\mathsf{Sign}}\langle x \mapsto +\rangle = \langle x \mapsto +\rangle,$$
$$[\![x := x - 1]\!]^{\mathsf{Sign}}\langle x \mapsto +\rangle = \langle x \mapsto \mathbb{Z}\rangle.$$

Hence, $[\![P]\!]^{\mathsf{Sign}}\alpha_{\mathsf{Sign}}(\{\langle x \mapsto v\rangle \mid v \in \mathbb{Z}\}) = \langle x \mapsto \mathbb{Z}\rangle$. We know that $\delta^{\mathfrak{w}}_{\mathsf{Sign}}(+, \mathbb{Z}) = 1$. Consequently, $P \in \mathbb{C}((\mathsf{Sign}, \delta^{\mathfrak{w}}_{\mathsf{Sign}}), 1)$, while $P \notin \mathbb{C}(\mathsf{Sign})$. ■

Similarly to the completeness case, for all ε, the partial completeness property is infinite and non-extensional. It is infinite because any number of sequential compositions of the empty command **skip** $\in \mathsf{Prog}$ is complete for all A, and, by Proposition 4.12, we know that they are also partially complete. Therefore, for all $\varepsilon \in \mathbb{Q}_{\geq 0}$: $|\mathbb{C}(\boldsymbol{A}, \varepsilon)| = \omega$. It is also non-extensional because there always exist programs P and Q such that: P is partially complete for A, $[\![P]\!] = [\![Q]\!]$, and Q is not partially complete for A. The following example shows the lack of extensionality in a simple partial completeness class.

Example 4.14. Consider the abstract domain $\mathsf{P} \sqcap \mathsf{S}$, shown in Figure 1.3. Let us consider the weighted path-length quasi-metric $\mathsf{P} \sqcap \mathsf{S}$-compatible $\delta^{\mathfrak{w}}_{\mathsf{P}\sqcap\mathsf{S}}$, and the partial completeness class $\mathbb{C}((\mathsf{P} \sqcap \mathsf{S}, \delta^{\mathfrak{w}}_{\mathsf{P}\sqcap\mathsf{S}}), 2)$. Consider the two programs P and Q in Figures 1.1, 1.2. As noted in Section 1, $[\![P]\!] = [\![Q]\!] = \lambda S \in \wp^{\mathrm{re}}(\mathbb{Z}).\{\langle x \mapsto 0\rangle\}$ and $\alpha_{\mathsf{P}\sqcap\mathsf{S}}([\![P]\!]S) = \alpha_{\mathsf{P}\sqcap\mathsf{S}}([\![Q]\!]S) = \langle x \mapsto 0\rangle$. It is easy to observe that

$$[\![P]\!]^{\mathsf{P}\sqcap\mathsf{S}}\alpha_{\mathsf{P}\sqcap\mathsf{S}}(\{\langle x \mapsto v\rangle \mid v \in \mathbb{Z}\}) = \langle x \mapsto \mathit{even}\rangle$$

while

$$[\![Q]\!]^{\mathsf{P}\sqcap\mathsf{S}}\alpha_{\mathsf{P}\sqcap\mathsf{S}}(\{\langle x \mapsto v\rangle \mid v \in \mathbb{Z}\}) = \langle x \mapsto \mathbb{Z}\rangle.$$

Consequently, $P \in \mathbb{C}((\mathsf{P} \sqcap \mathsf{S}, \delta^{\mathfrak{w}}_{\mathsf{P}\sqcap\mathsf{S}}), 2)$ while $Q \notin \mathbb{C}((\mathsf{P} \sqcap \mathsf{S}, \delta^{\mathfrak{w}}_{\mathsf{P}\sqcap\mathsf{S}}), 2)$. Note that, even though P is partially complete w.r.t. $\mathbb{C}((\mathsf{P} \sqcap \mathsf{S}, \delta^{\mathfrak{w}}_{\mathsf{P}\sqcap\mathsf{S}}), 2)$, both P and Q are not complete for $\mathsf{P} \sqcap \mathsf{S}$, i.e., $P, Q \notin \mathbb{C}(\mathsf{P} \sqcap \mathsf{S})$. ■

4.3 Classes of Partial Complete Programs

It is well known that for trivial abstractions the corresponding completeness class turns out to be the whole set of programs [43]. Moreover, for all non-trivial abstractions in $\mathscr{A}(\wp^{\mathrm{re}}(\mathbb{S}))$, its completeness class is strictly contained in

Prog [44]. More has been recently proved along this direction: a completeness class is an index set of partial recursive function if and only if the abstraction is trivial [8]. In this section, we study the counterpart of these results for the case of partial completeness. We first consider the simplest case of abstract quasi-metric spaces of stores $\boldsymbol{A} \in \mathfrak{A}(\wp^{\mathrm{re}}(\mathbb{S}))$ satisfying the property of having a limited imprecision, i.e., complete lattices A such that the quasi-metric δ_A is bounded. These include, for instance, the case of finite height lattices with the weighted path-length quasi-metric—complete lattices which are both ACC and DCC. Although not very expressive, these abstract domains are widely used in Galois insertion-based abstract interpretations to enforce termination of the analysis. We then consider the more general setting of abstract interpretations over abstract domains where an unlimited imprecision can always be produced by any terminating program analysis by abstract interpretation, e.g., employing widening operations to enforce termination [23, 26]. This will include the most general case of abstract interpretation.

Definition 4.15 (Abstract quasi-metric spaces with limited imprecision). *An abstract quasi-metric space of stores $\boldsymbol{A} \in \mathfrak{A}(\wp^{\mathrm{re}}(\mathbb{S}))$ has imprecision limited by $\varepsilon \in \mathbb{Q}_{\geq 0}$ if for each $a \in A$ we have $\delta_A(\bot_A, a) \leq \varepsilon$.*

The following result is immediate and helps us to understand the relation between abstract domains with limited imprecision and the class of partial completeness properties of programs.

Proposition 4.16. *If $\boldsymbol{A} \in \mathfrak{A}(\wp^{\mathrm{re}}(\mathbb{S}))$ has limited imprecision, then:*

$$\exists \varepsilon \in \mathbb{Q}_{\geq 0}.\ \mathbb{C}(\boldsymbol{A}, \varepsilon) = \mathsf{Prog}\,.$$

Proof. By definition of abstract domain with limited imprecision, there exists $\varepsilon \in \mathbb{Q}_{\geq 0}$ such that if $a \in A$ then $\delta_A(\bot_A, a) \leq \varepsilon$. Consider $n \in \mathbb{Q}_{\geq 0}$ such that $n \geq \varepsilon$. Then $\mathsf{Prog} = \mathbb{C}(\boldsymbol{A}, \infty) = \cup_{m \leq n} \mathbb{C}(\boldsymbol{A}, m)$. □

The difference with respect to the case of standard completeness class $\mathbb{C}(A)$, i.e., the set of all programs that are complete for the abstract domain A, is that, thanks to the possibility of admitting an upper margin to imprecision (i.e., possible false alarms), then there always exists a class of partial completeness with respect to a given bound which includes all programs. This corresponds to allow the largest possible imprecision, viz., amount of incompleteness, making the condition of being complete (viz., precise) vacuous.

Example 4.17. Consider the abstract quasi-metric space $(\mathsf{Sign}, \delta^{\mathfrak{w}}_{\mathsf{Sign}}) \in \mathfrak{A}(\wp^{\mathrm{re}}(\mathbb{Z}))$. $(\mathsf{Sign}, \delta^{\mathfrak{w}}_{\mathsf{Sign}})$ has clearly limited imprecision. Indeed, it is easy to note that for every $n \geq 3$, $\mathbb{C}((\mathsf{Sign}, \delta^{\mathfrak{w}}_{\mathsf{Sign}}), n) = \mathsf{Prog}$. ■

The case of abstract domains with unlimited imprecision is less straightforward and reflects precisely, into the theory of partial completeness, a similar result holding the completeness case [44]. An abstract quasi-metric space of stores $\boldsymbol{A} \in \mathfrak{A}(\wp^{\mathrm{re}}(\mathbb{S}))$ has *unlimited imprecision* when:

$$\forall \varepsilon \in \mathbb{Q}_{\geq 0}.\ \exists a \in A.\ \delta_A(\bot_A, a) > \varepsilon.$$

Clearly, the unlimited imprecision property can only be satisfied by abstract quasi-metric space of stores with a (countably) infinite number of elements.

Proposition 4.18. *Let $\boldsymbol{A} \in \mathfrak{A}(\wp^{\mathrm{re}}(\mathbb{S}))$ be an abstract quasi-metric space of stores with unlimited imprecision. Then $|A| = \omega$.*

Proof. Immediate by definition. □

Theorem 4.19. *Let $\boldsymbol{A} \in \mathfrak{A}(\wp^{\mathrm{re}}(\mathbb{S}))$ be some abstract quasi-metric space of stores with unlimited imprecision for some A-compatible quasi-metric δ_A. Then:*

$$\exists \varepsilon \in \mathbb{Q}_{\geq 0}.\ \mathbb{C}(\boldsymbol{A}, \varepsilon) = \mathsf{Prog} \ \Leftrightarrow\ A = id.$$

Proof. ($\Leftarrow$) is straightforward because when $A = id$ then $\mathbb{C}(\boldsymbol{A}, 0) = \mathbb{C}(A) = \mathsf{Prog}$.
($\Rightarrow$) The proof is based on the construction of an opaque predicate [18], i.e., a predicate that when executed by the concrete interpreter on some set of input stores S is always false but, when executed by the abstract interpreter on the abstraction of S it may result true for some stores in S and false for other stores in S. This generalizes to the case of partial completeness a result in [44] for standard completeness.

By contradiction, assume that:

(1) $A \neq id$ and, by Assumption 4, A is a strict abstract domain, i.e., with abstraction and concretization functions, respectively α_A and γ_A, such that $\gamma_A(\alpha_A(\varnothing)) = \varnothing$;

(2) $\boldsymbol{A}$ has unlimited imprecision, i.e., $\forall \varepsilon \in \mathbb{Q}_{\geq 0}.\ \exists a \in A.\ \delta_A(\bot_A, a) > \varepsilon$;

(3) there exists $\xi \geq 0$ such that $\mathbb{C}(\boldsymbol{A}, \xi) = \mathsf{Prog}$.

By (2) and (3) we have that there exists $a \in A$ such that $\bot_A \neq a$, otherwise if $\bot_A = a$ then $\delta_A(\bot_A, a) = 0$, while we have $\delta_A(\bot_A, a) > \xi$ with $\xi \in \mathbb{Q}_{\geq 0}$. Hence by (1), $\varnothing \subset \gamma_A(a)$ (recall that in any Galois insertion γ_A is injective). By (1) there exists $S \in \wp^{\mathrm{re}}(\mathbb{S})$ such that $S \neq \varnothing$ (because A is strict) and $S \subset \gamma_A(\alpha_A(S))$ (because $\gamma_A \circ \alpha_A \neq id$).

We are now in the position of building a program that does not belong to $\mathbb{C}(\boldsymbol{A}, \xi)$, therefore leading to an absurd. By (1) an opaque predicate can be built for our abstract domain A. Let $z \in \gamma_A(\alpha_A(S)) \smallsetminus S$. Because z is a (finite

variable) store, then we can build a boolean predicate $\mathtt{Is}(z) \in \mathrm{BExp}$ such that for all $s \in \mathbb{S}$:

$$(\!|\mathtt{Is}(z)|\!)s = \mathbf{t} \;\Leftrightarrow\; z = s.$$

In this case we have that

$$\begin{aligned} [\![\mathtt{Is}(z)]\!]S &= \varnothing \\ [\![\neg\mathtt{Is}(z)]\!]S &= S \\ \gamma_A([\![\mathtt{Is}(z)]\!]^A\alpha_A(S)) &\neq \varnothing. \end{aligned}$$

Define $P_a \in \mathsf{Prog}$ as a program such that for every store $s \in \mathbb{S}$ we have that $[\![P_a]\!]s \in \gamma_A(a)$ and in particular $[\![P_a]\!]^A([\![\mathtt{Is}(z)]\!]^A\alpha_A(S)) = a$. Turing completeness of Prog ensures that we can compute any finite store in $\gamma_A(a)$. Moreover, being $\gamma_A(a)$ and $\gamma_A(\alpha_A(S))$ both recursive sets, and $[\![\cdot]\!]^A$ a program, then we can build P_a in Prog as above. Consider also $P_w \triangleq \mathbf{while\ t\ do\ skip}$. In this latter case, we would have that for every store $s \in \mathbb{S}$: $[\![P_w]\!]s = \varnothing$ and therefore for all $X \subseteq \mathbb{S}$: $[\![P_w]\!]^A\alpha_A(X) = \bot_A$. We prove that if

$$P \triangleq \mathbf{if}\ \mathtt{Is}(z)\ \mathbf{then}\ P_a\ \mathbf{else}\ P_w$$

then $\delta_A(\alpha_A([\![P]\!]S), [\![P]\!]^A\alpha_A(S)) > \xi$ hence $P \notin \mathbb{C}(\boldsymbol{A}, \xi)$. It is clear that

$$\begin{aligned} [\![P]\!]S = \\ [\![P_a]\!]([\![\mathtt{Is}(z)]\!]S) \cup [\![P_w]\!]([\![\neg\mathtt{Is}(z)]\!]S) = \\ [\![P_a]\!](\varnothing) \cup \varnothing = \\ \varnothing \end{aligned}$$

and therefore $\alpha_A([\![P]\!]S) = \bot_A$. Analogously, because for every set of stores $X \subseteq \mathbb{S}$: $[\![P_w]\!]^A\alpha_A(X) = \bot_A$, we have:

$$\begin{aligned} [\![P]\!]^A\alpha_A(S) = \\ [\![P_w]\!]^A([\![\neg\mathtt{Is}(z)]\!]^A\alpha_A(S)) \sqcup_A [\![P_a]\!]^A([\![\mathtt{Is}(z)]\!]^A\alpha_A(S)) = \\ \bot_A \sqcup_A a = \\ a \end{aligned}$$

Therefore: $\delta_A(\alpha_A([\![P]\!]S), [\![P]\!]^A\alpha_A(S)) = \delta_A(\bot_A, a) > \xi$, hence, $P \notin \mathbb{C}(\boldsymbol{A}, \xi)$ which contradicts assumption (3). We can conclude that A must be the identical abstraction of stores. □

Informally, if we consider a non-trivial abstract quasi-metric space that has unlimited imprecision, then independently of how we set a threshold ε of false alarms acceptance, namely $\mathbb{C}(\boldsymbol{A}, \varepsilon)$, there always exists a program P for which the abstract analysis over A, is not ε-partially complete, namely $P \notin \mathbb{C}(\boldsymbol{A}, \varepsilon)$. By a straightforward padding argument, any of these programs can be extended

to an infinite set of programs for which the abstraction is ε-*partially incomplete*. The class of ε-partial *incomplete* programs for $\boldsymbol{A} \in \mathfrak{A}(\wp^{\mathrm{re}}(\mathbb{S}))$, is the complement set of $\mathbb{C}(\boldsymbol{A}, \varepsilon)$, formally:

$$\overline{\mathbb{C}(\boldsymbol{A}, \varepsilon)} \triangleq \{P \in \mathsf{Prog} \mid \exists S \in \wp^{\mathrm{re}}(\mathbb{S}).\ \alpha_A([\![P]\!]S) \not\approx^{\varepsilon}_{\delta_A} [\![P]\!]^A \alpha_A(S)\}.$$

Corollary 4.20. *If $\boldsymbol{A}$ is a non-trivial abstract quasi-metric space with unlimited imprecision, then for all $\varepsilon \in \mathbb{Q}_{\geq 0}$, $|\overline{\mathbb{C}(\boldsymbol{A}, \varepsilon)}| = \omega$.*

Proof. $\boldsymbol{A}$ is not trivial and has unlimited imprecision therefore, by Theorem 4.19, for every $\varepsilon \in \mathbb{Q}_{\geq 0}$ we have

$$\mathbb{C}(\boldsymbol{A}, \varepsilon) \neq \mathsf{Prog} \ \Leftrightarrow\ \overline{\mathbb{C}(\boldsymbol{A}, \varepsilon)} \neq \mathsf{Prog}\,.$$

By padding all programs in $\overline{\mathbb{C}(\boldsymbol{A}, \varepsilon)}$ with, e.g., **skip** instructions we get the expected result $|\overline{\mathbb{C}(\boldsymbol{A}, \varepsilon)}| = \omega$. □

This means that, any non-trivial abstract domain of stores endowed with an unlimited quasi-metric δ_A, has an infinite set of programs for which the abstract interpreter is ε-partially incomplete. Conversely, if δ_A has limited imprecision, then, trivially, we can always find a certain level of tolerance that makes the analysis ε-partially complete for all programs.

Example 4.21. Consider the abstract domain of intervals $\mathsf{Int} \in \mathscr{A}(\wp^{\mathrm{re}}(\mathbb{Z}))$, shown in Example 3.3, endowed with a quasi-metric Int-compatible δ_{Int}, such that δ_{Int} has unlimited imprecision. Recall that Int is not ACC. This means that there are infinite strictly ascending chains, such as for instance $[0,1] \leq_{\mathsf{Int}} [0,2] \leq_{\mathsf{Int}} \cdots \leq_{\mathsf{Int}} [0,n], \ldots$. Hence, a proper widening operator is required in order to enforce convergence of the abstract Kleene iterates of the abstract interpreter:

$$\lambda X^\sharp.\ S^\sharp \sqcup_A [\![C]\!]^A [\![b]\!]^A X^\sharp.$$

The standard interval widening consists in replacing any unstable upper bound with $+\infty$ and any unstable lower bound with $-\infty$. Let us define in Figure 4.2 the "delayed" widening $\nabla^i_{\mathsf{Int}} : \mathsf{Int} \times \mathsf{Int} \to \mathsf{Int}$, where $\#_{iter}$ indicates the current number of iteration in the loop. That is, ∇^i_{Int} does not immediately abort unstable computations by pushing to infinity, but it delays its application after i iterations. This is particularly useful when the first few iterates of the loop differ from the following ones, and it is always a good idea to start extrapolating only after having accumulated a few iterations. In this way, the widening can make a more educated guess about the loop behavior. After a finite, fixed number of i iterations, we revert to widening so that termination of the abstract interpreter is preserved.

Consider the following program P_n :

$$[a,b]\ \nabla^i_{\mathsf{Int}}\ [c,d] \triangleq \left[\begin{cases} a & \text{if } a \leq c \\ c & \text{if } c < a \text{ and } \#_{iter} \leq i \\ -\infty & \text{otherwise} \end{cases}, \begin{cases} b & \text{if } b \geq d \\ d & \text{if } d > b \text{ and } \#_{iter} \leq i \\ +\infty & \text{otherwise} \end{cases}\right]$$

Figure 4.2: The widening operator in Example 4.21

$$\begin{array}{l} x := n; \\ \textbf{while } x > 1 \textbf{ do} \\ \qquad x := x - (n-1) \end{array}$$

where $n \in \mathbb{N}_{\geq 1}$ is a constant and the expressions n and $(n-1)$ are assumed to be resolved before giving the program as input to the concrete and abstract interpreter (e.g., by running a specializer on the code like the pre-processor for the C language which replaces all the occurrences of symbols n with the corresponding constant number). Clearly, for every input store the while-loop of P_n terminates after one iteration if the condition $x > 1$ is satisfied. Indeed, for all $n \geq 1$ and $S \in \wp^{\mathrm{re}}(\mathbb{Z})$, we have $[\![P_n]\!]S = \{\langle x \mapsto 1\rangle\}$, which corresponds to

$$\alpha_{\mathsf{Int}}([\![P_n]\!]S) = \alpha_{\mathsf{Int}}(\{\langle x \mapsto 1\rangle\}) = [1,1].$$

This means that all P_n are extensionally equivalent. Consider now the abstract denotational semantics for Int abstract domain of stores, where we replace the abstract lub $\sqcup_{\mathsf{Int}}$ in the loop head with the widening ∇^2_{Int} which forces the convergence after two iterations. If $n = 1$ we get trivially $\langle x \mapsto [1,1]\rangle$. For $n \geq 2$, at the first iteration of the while-loop we have:

$$\begin{array}{rl} [\![x > 1]\!]^{\mathsf{Int}}\langle x \mapsto [n,n]\rangle &= \langle x \mapsto [n,n]\rangle \\ [\![x := x - (n-1)]\!]^{\mathsf{Int}}\langle x \mapsto [n,n]\rangle &= \langle x \mapsto [1,1]\rangle \\ \langle x \mapsto [n,n]\rangle\ \nabla^2_{\mathsf{Int}}\ \langle x \mapsto [1,1]\rangle &= \langle x \mapsto [1,n]\rangle \end{array}$$

while for the second iteration:

$$\begin{array}{rl} [\![x > 1]\!]^{\mathsf{Int}}\langle x \mapsto [1,n]\rangle &= \langle x \mapsto [2,n]\rangle \\ [\![x := x - (n-1)]\!]^{\mathsf{Int}}\langle x \mapsto [2,n]\rangle &= \langle x \mapsto [(3-n),1]\rangle \\ \langle x \mapsto [1,n]\rangle\ \nabla^2_{\mathsf{Int}}\ \langle x \mapsto [(3-n),1]\rangle &= \langle x \mapsto [(3-n),n]\rangle. \end{array}$$

Finally, at the third iteration we get a fixpoint. The final abstract output of the abstract semantics is given by the lub between

$$[\![x \leq 1]\!]^{\mathsf{Int}}\langle x \mapsto [1,n]\rangle \sqcup_{\mathsf{Int}} [\![x \leq 1]\!]^{\mathsf{Int}}\langle x \mapsto [(3-n),n]\rangle$$

which is $\langle x \mapsto [(3-n),1]\rangle$. Therefore, the abstract semantics of P_n for every input $S \in \wp^{\mathrm{re}}(\mathbb{Z})$ is

$$[\![P_1]\!]^{\mathsf{Int}}\alpha_{\mathsf{Int}}(S) = \langle x \mapsto [1,1]\rangle$$
$$[\![P_{n\geq 2}]\!]^{\mathsf{Int}}\alpha_{\mathsf{Int}}(S) = \langle x \mapsto [(3-n),1]\rangle.$$

Note that $\{P_n\}_{n\in\{1,2\}} \subset \mathbb{C}(\mathsf{Int})$ while $\{P_n\}_{n>2} \cap \mathbb{C}(\mathsf{Int}) = \varnothing$. That is, all programs in $\{P_n\}_{n>2}$ are incomplete for Int. Moreover, we can arbitrarily worsen the result of our static analysis on P_n by selecting a larger constant n. This implies that, since δ_{Int} has unlimited imprecision by assumption, if $P_n \in \mathbb{C}((\mathsf{Int}, \delta_{\mathsf{Int}}), \varepsilon)$ for some $n > 2$ and $\varepsilon \in \mathbb{Q}_{\geq 0}$, then there exists a constant $m \in \mathbb{N}_{>0}$ such that, for all $S \in \wp^{\mathrm{re}}(\mathbb{Z})$, the output of the abstract interpreter is

$$[\![P_{n+m}]\!]^{\mathsf{Int}}\alpha_{\mathsf{Int}}(S) = \langle x \mapsto [(3-(n+m)),1]\rangle$$

and $\langle x \mapsto [(3-n),1]\rangle$ is not ε-close to $\langle x \mapsto [(3-(n+m)),1]\rangle$, namely

$$\langle x \mapsto [(3-n),1]\rangle \not\approx^{\varepsilon}_{\delta_{\mathsf{Int}}} \langle x \mapsto [(3-(n+m)),1]\rangle.$$

This implies that $P_{n+m} \notin \mathbb{C}((\mathsf{Int}, \delta_{\mathsf{Int}}), \varepsilon)$. Observe that, even though both $x > 1$ and $x \leq 1$ are exactly representable in Int with the intervals, respectively, $[2, +\infty]$ and $[-\infty, 1]$, the transfer function for $[\![x \leq 1]\!] : \wp^{\mathrm{re}}(\mathbb{Z}) \to \wp^{\mathrm{re}}(\mathbb{Z})$ is incomplete with respect to Int. This imprecision can be arbitrary widened, without modifying the extensional behavior of the program. This makes possible to foil any partial complete abstraction with respect to any constant bound $\varepsilon \in \mathbb{Q}_{\geq 0}$. ■

We now focus on the investigation of the computational limits of the class of partially complete and incomplete programs with respect to a given abstract quasi-metric space and ε bound. We begin with the definition of ε-triviality for an abstract quasi-metric space of stores.

Definition 4.22 (ε-trivial Abstract quasi-metric spaces). *An abstract quasi-metric space of stores $\boldsymbol{A} \in \mathfrak{A}(\wp^{\mathrm{re}}(\mathbb{S}))$ is ε-trivial for some $\varepsilon \in \mathbb{Q}_{\geq 0}$ if $\mathbb{C}(\boldsymbol{A}, \varepsilon) = \mathsf{Prog}$.*

Of course ε-trivial abstract domains induce recursive classes of partial complete programs. Indeed, if $\boldsymbol{A}$ is ε-trivial, then both $\mathbb{C}(\boldsymbol{A}, \varepsilon) = \mathsf{Prog}$ and $\overline{\mathbb{C}(\boldsymbol{A}, \varepsilon)} = \varnothing$ are recursive sets. The concept of ε-triviality extends naturally the concept of trivial abstract domain of stores, as highlighted in the following proposition.

Proposition 4.23. *For all $\boldsymbol{A} \in \mathfrak{A}(\wp^{\mathrm{re}}(\mathbb{S}))$ and $\varepsilon \in \mathbb{Q}_{\geq 0}$, the following hold:*

(i) $A \in \{id, \top^{\mathbb{S}}\} \Rightarrow \boldsymbol{A}$ is ε-trivial for all $\varepsilon \geq 0$;

(ii) $\boldsymbol{A}$ *limited imprecision* $\Rightarrow \exists \varepsilon \in \mathbb{Q}_{\geq 0}$. $\boldsymbol{A}$ *is* ε*-trivial;*

(iii) $A \neq id$ *and* $\boldsymbol{A}$ *unlimited imprecision* $\Rightarrow$ $\boldsymbol{A}$ *is not* ε*-trivial for all* $\varepsilon \geq 0$*;*

Proof. (i) follows by $\mathbb{C}(id) = \mathbb{C}(\top^{\mathbb{S}}) = \mathbb{C}((id, \delta_{id}), \varepsilon) = \mathbb{C}((\top^{\mathbb{S}}, \delta_{\top^{\mathbb{S}}}), \varepsilon) = \mathsf{Prog}$ for all $\varepsilon \in \mathbb{Q}_{\geq 0}$; (ii) immediate by Proposition 4.16; (iii) by Corollary 4.20 $A \neq \top^{\mathbb{S}}$ therefore, by Theorem 4.19, $\boldsymbol{A}$ is not ε-trivial for all $\varepsilon \geq 0$. □

In the following, we show some simple examples of ε-trivial abstract domains.

Example 4.24. $(id, \delta_{id}^{\mathfrak{w}})$ and $(\top^{\mathbb{S}}, \delta_{\top^{\mathbb{S}}}^{\mathfrak{w}})$ are both n-trivial for all $n \in \mathbb{Q}_{\geq 0}$. The $(\mathsf{Sign}, \delta_{\mathsf{Sign}}^{\mathfrak{w}})$ abstract quasi-metric space is n-trivial for $n \geq 3$. $(\mathsf{Parity}, \delta_{\mathsf{Parity}}^{\mathfrak{w}})$ is n-trivial for $n \geq 2$, while $(\mathsf{P} \sqcap \mathsf{S}, \delta_{\mathsf{P} \sqcap \mathsf{S}}^{\mathfrak{w}})$ is n-trivial for $n \geq 4$. ■

The following theorems prove that both $\mathbb{C}(\boldsymbol{A}, \varepsilon)$ and $\overline{\mathbb{C}(\boldsymbol{A}, \varepsilon)}$ are non-recursively enumerable sets. Moreover, we show that the class of ε-partial incomplete programs $\overline{\mathbb{C}(\boldsymbol{A}, \varepsilon)}$ plays a special role between all the non-r.e. sets: it is a *productive* set.

Theorem 4.25. *Let* $\boldsymbol{A} \in \mathfrak{A}(\wp^{\mathrm{re}}(\mathbb{S}))$ *be an abstract quasi-metric space. If* $\boldsymbol{A}$ *is not* ε*-trivial, then, for all* $\varepsilon \in \mathbb{Q}_{\geq 0}$, $\mathbb{C}(\boldsymbol{A}, \varepsilon)$ *and* $\overline{\mathbb{C}(\boldsymbol{A}, \varepsilon)}$ *are not recursively enumerable.*

Proof. The proof is made by showing that the first order predicate defining the class of ε-partially complete (resp. incomplete) programs (which can be seen as a subset of natural numbers since each program in Prog can be mapped through the Gödel numbering $\mathfrak{g} : \mathsf{Prog} \to \mathbb{N}$ to a natural number) is in Π_2 (resp. Σ_2). Let $\varphi_P : \mathbb{S} \nrightarrow \mathbb{S}$ be the partial recursive function associated to program $P \in \mathsf{Prog}$ (the concrete interpreter) and, without loss of generality, $\varphi_A : \mathsf{Prog} \times A \to A$ be the total recursive function representing the abstract interpreter. We define the predicate $R(\boldsymbol{A}, \varepsilon, P, s, n)$, with $s \in \mathbb{S}$ and $n \in \mathbb{N}$, as follows:

$$R(\boldsymbol{A}, \varepsilon, P, s, n) \Leftrightarrow \varphi_P(s)\downarrow \text{ in } n \text{ steps } \wedge\ \delta_A(\alpha_A(\varphi_P(s)), \varphi_A(P, \alpha_A(s))) \leq \varepsilon.$$

Because $\delta_A(a, b) \leq \varepsilon$ is a decidable predicate by definition of δ_A, the predicate defining R is recursive, i.e.:

$$\varphi_P(s)\downarrow \text{ in } n \text{ steps } \wedge\ \delta_A(\alpha_A(\varphi_P(s)), \varphi_A(P, \alpha_A(s))) \leq \varepsilon\ \in\ \Sigma_0 = \Pi_0.$$

It follows that R is recursive too, i.e., $R(\boldsymbol{A}, \varepsilon, P, s, n) \in \Sigma_0 = \Pi_0$. We can rewrite the decidability of the ε-partial completeness and incompleteness classes as follows:

$$P \in \mathbb{C}(\boldsymbol{A}, \varepsilon) \;\Leftrightarrow\; \forall s \in \mathbb{S}\,.\,\exists n \in \mathbb{N}.\; R(\boldsymbol{A}, \varepsilon, P, s, n)$$
$$P \in \overline{\mathbb{C}(\boldsymbol{A}, \varepsilon)} \;\Leftrightarrow\; \exists s \in \mathbb{S}\,.\,\forall n \in \mathbb{N}.\; \neg R(\boldsymbol{A}, \varepsilon, P, s, n).$$

By observing the bounded quantifiers before $R(\boldsymbol{A}, \varepsilon, P, s, n)$ and $\neg R(\boldsymbol{A}, \varepsilon, P, s, n)$, we can conclude that

$$\forall s \in \mathbb{S}\,.\,\exists n \in \mathbb{N}.\; R(\boldsymbol{A}, \varepsilon, P, s, n) \in \Pi_2$$
$$\exists s \in \mathbb{S}\,.\,\forall n \in \mathbb{N}.\; \neg R(\boldsymbol{A}, \varepsilon, P, s, n) \in \Sigma_2.$$

This proves that $\mathbb{C}(\boldsymbol{A}, \varepsilon)$ and $\overline{\mathbb{C}(\boldsymbol{A}, \varepsilon)}$ are both non-r.e. sets. □

Corollary 4.26. *For every non-trivial $A \in \mathscr{A}(\wp^{\mathrm{re}}(\mathbb{S}))$, the classes of complete $\mathbb{C}(A)$ and incomplete programs $\overline{\mathbb{C}(A)}$ are not recursively enumerable.*

Proof. Recall that the equivalence relation in A is decidable by Definition 3.9, indeed for all $a, b \in A$, $a = b \;\Leftrightarrow\; a \preceq_A b \wedge b \preceq_A a$. The proof is straightforward from the proof of Theorem 4.25 by setting

$$R(A, P, s, n) \;\Leftrightarrow\; \varphi_P(s)\!\downarrow \text{ in } n \text{ steps} \;\wedge\; \alpha_A(\varphi_P(s)) = \varphi_A(P, \alpha_A(s)).$$

□

Theorem 4.27. *Let $\boldsymbol{A} \in \mathfrak{A}(\wp^{\mathrm{re}}(\mathbb{S}))$ be some non-trivial abstract quasi-metric space with unlimited imprecision. Then, for every $\varepsilon \in \mathbb{Q}_{\geq 0}$, $\overline{\mathbb{C}(\boldsymbol{A}, \varepsilon)}$ is productive.*

Proof. In the following, without loss of generality, we assume that programs in Prog have a single variable ranging over $\mathbb{N}$, so that $\mathbb{S} = \mathbb{N}$. Let $\mathfrak{g} : \mathsf{Prog} \to \mathbb{N}$ be the Gödel numbering of all programs in Prog. In order to prove that $\overline{\mathbb{C}(\boldsymbol{A}, \varepsilon)}$ is productive, we need a many-to-one reduction from a productive set X to $\overline{\mathbb{C}(\boldsymbol{A}, \varepsilon)}$, namely, $X \preceq_f \overline{\mathbb{C}(\boldsymbol{A}, \varepsilon)}$, by the application of a total recursive function $f : \mathsf{Prog} \to \mathsf{Prog}$ (f is a program transformer) such that $P \in X$ iff $f(P) \in \overline{\mathbb{C}(\boldsymbol{A}, \varepsilon)}$. Consider the representation in Prog of the halting problem of Turing machines (cf. [68]):

$$K \triangleq \{P \in \mathsf{Prog} \mid [\![P]\!]\{\mathfrak{g}(P)\} \neq \varnothing\}.$$

It is well known that K is r.e. while $\overline{K}$ is productive [68]. We prove the productivity of $\overline{\mathbb{C}(\boldsymbol{A}, \varepsilon)}$ by showing $\overline{K} \preceq_f \overline{\mathbb{C}(\boldsymbol{A}, \varepsilon)}$ for all $\varepsilon \geq 0$. Recall that

$$\overline{K} \preceq_f \overline{\mathbb{C}(\boldsymbol{A}, \varepsilon)} \;\Leftrightarrow\; K \preceq_f \mathbb{C}(\boldsymbol{A}, \varepsilon)$$

we construct a program transformer $f : \mathsf{Prog} \to \mathsf{Prog}$, i.e., a total recursive function, such that for all $P \in \mathsf{Prog}$:

$$P \in K \;\Leftrightarrow\; f(P) \in \mathbb{C}(\boldsymbol{A}, \varepsilon).$$

By assumptions, we know that:

(1) $\boldsymbol{A}$ is not trivial, namely, $A \notin \{id, \top^{\mathbb{N}}\}$,

(2) by Proposition 3.11, $\gamma_A(\alpha_A(\varnothing)) = \varnothing$, and

(3) $\boldsymbol{A}$ has unlimited imprecision.

We build a predicate $\mathtt{Is}(q) \in \mathrm{BExp}$ with $q \in \mathbb{N}$ such that, as done in the proof of Theorem 4.19, for all $s \in \mathbb{N}$, $(\!|\mathtt{Is}(q)|\!)s = \mathbf{t} \Leftrightarrow q = s$. Let us consider two sets $Q, R \subseteq \mathbb{N}$ and a store $z \in \mathbb{N}$ such that:

- $\alpha_A(\varnothing) \not\approx^{\varepsilon}_{\delta_A} \alpha_A(Q)$ and we know that such set of stores Q exists thanks to assumption (3);
- $R \neq \varnothing$, $R \subset \gamma_A(\alpha_A(R))$, $z \in \gamma_A(\alpha_A(R)) \smallsetminus R$ and $[\![\mathtt{Is}(z)]\!]^A\alpha_A(R) = a$ such that $\alpha_A(\varnothing) \not\approx^{\varepsilon}_{\delta_A} a$. The set of stores R exists thanks to assumptions (1), (2) and (3). Indeed, for non-trivial abstractions of stores, there always exists an element in $c \in \wp^{\mathrm{re}}(\mathbb{N})$ that is not perfectly represented in A, i.e., $c \subset \gamma_A(\alpha_A(c))$ and, by assuming that $\boldsymbol{A}$ has unlimited imprecision, we can choose an element $b \in A$ such that the abstraction of c is b and $\alpha_A(\varnothing) \not\approx^{\varepsilon}_{\delta_A} b$. This reasoning can be extended w.r.t. the request $[\![\mathtt{Is}(z)]\!]^A\alpha_A(R) = a$.

Consider also the never terminating program $P_w \triangleq \textbf{while t do skip}$. Let us assume that we can evaluate (in a "call by value" evaluation) programs in boolean guards and such calls are denoted by the classical semantic notation $[\![F]\!]\{s_{\mathrm{in}}\}$, where $F \in \mathsf{Prog}$ and $s_{\mathrm{in}} \in \mathbb{N}$. A boolean guard having the form $[\![F]\!]\{s_{\mathrm{in}}\} \neq \varnothing \in \mathrm{BExp}$ is true whenever program F terminates on input s_{in}, namely, for all $s, s_{\mathrm{in}} \in \mathbb{N}$:

$$(\!|[\![P]\!]\{s_{\mathrm{in}}\} \neq \varnothing|\!)s = \mathbf{t} \Leftrightarrow [\![P]\!]\{s_{\mathrm{in}}\} \neq \varnothing.$$

We define for any $P \in \mathsf{Prog}$ the program transformer $f : \mathsf{Prog} \to \mathsf{Prog}$ as follows:

$$f(P) \triangleq \textbf{if } (\mathtt{Is}(z) \vee [\![P]\!]\{\mathfrak{g}(P)\} \neq \varnothing) \textbf{ then skip else } P_w$$

Suppose that $P \in K$. This implies that for every store $s \in \mathbb{N}$:

$$(\!|[\![P]\!]\{\mathfrak{g}(P)\} \neq \varnothing|\!)s = \mathbf{t}.$$

Therefore, for all sets of stores $S \in \wp^{\mathrm{re}}(\mathbb{N})$:

$$\begin{aligned}
&\alpha_A([\![f(P)]\!]S) = \\
&\alpha_A([\![\textbf{skip}]\!]([\![\mathtt{Is}(z) \vee [\![P]\!]\{\mathfrak{g}(P)\} \neq \varnothing]\!]S) \cup \\
&[\![P_\omega]\!]([\![\neg\mathtt{Is}(z) \wedge [\![P]\!]\{\mathfrak{g}(P)\} = \varnothing]\!]S)) = \\
&\alpha_A([\![\textbf{skip}]\!]S \cup [\![P_\omega]\!]\varnothing) = \\
&\alpha_A(S).
\end{aligned}$$

By Assumption 3, we have that

$$[\![P]\!]\{\mathfrak{g}(P)\} \neq \varnothing \;\Rightarrow\; [\![P]\!]^A\alpha_A(\{\mathfrak{g}(P)\}) \neq \alpha_A(\varnothing)$$

therefore, for all $S \in \wp^{\mathrm{re}}(\mathbb{N})$, we have the following abstract evaluations:

$$[\![f(P)]\!]^A\alpha_A(S) = [\![\mathbf{skip}]\!]^A\alpha_A(S) = \alpha_A(S).$$

Hence, we can conclude that program $f(P)$ is complete, thus (by Proposition 4.12) ε-partially complete for A, that is, $f(P) \in \mathbb{C}(A) = \mathbb{C}(\boldsymbol{A}, 0) \subseteq \mathbb{C}(\boldsymbol{A}, \varepsilon)$.

Suppose that $P \notin K$. We show that there exists a set of stores for which $f(P)$ is ε-partially incomplete for all $\varepsilon \geq 0$. Being P not in K, implies that the program $f(P)$ gets stuck on the evaluation of $[\![P]\!]\{\mathfrak{g}(P)\} \neq \varnothing$. This implies that, for all $S \in \wp^{\mathrm{re}}(\mathbb{N})$, $\alpha_A([\![f(P)]\!]S) = \alpha_A(\varnothing)$. By Assumption 3, we know that the abstract interpreter always evaluates in a finite number of steps the boolean guard $[\![P]\!]^A\alpha_A(\{\mathfrak{g}(P)\}) \neq \alpha_A(\varnothing)$. Therefore, we have two possibilities: the abstract interpreter is either complete (hence it has caught the non-termination of the specific program P with input $\mathfrak{g}(P)$ although it is not possible for all $P \in \mathsf{Prog}$) on the abstract evaluation of $[\![P]\!]^A\alpha_A(\{\mathfrak{g}(P)\})$, i.e., for all $S \in \wp^{\mathrm{re}}(\mathbb{N})$:

$$\alpha_A([\![[\![P]\!]\{\mathfrak{g}(P)\} \neq \varnothing]\!]S) \;=\; [\![[\![P]\!]^A\alpha_A(\{\mathfrak{g}(P)\}) \neq \alpha_A(\varnothing)]\!]^A\alpha_A(S) = \bot_A$$

or incomplete. Assume that A is complete for the specific program P on the check $[\![P]\!]\{\mathfrak{g}(P)\} \neq \varnothing$. This implies that the evaluation of the boolean guard of $f(P)$ is reduced to the evaluation of $\mathtt{Is}(z)$ only. By using the set of stores R defined before, we know that $\gamma_A([\![\mathtt{Is}(z)]\!]^A\alpha_A(R)) \neq \varnothing$. This leads to:

$$\begin{aligned}
[\![f(P)]\!]^A\alpha_A(R) = \\
[\![\mathbf{skip}]\!]^A([\![\mathtt{Is}(z) \vee [\![P]\!]\{\mathfrak{g}(P)\} \neq \varnothing]\!]^A\alpha_A(R)) \sqcup_A \\
[\![P_\omega]\!]^A([\![\neg\mathtt{Is}(z) \wedge [\![P]\!]\{\mathfrak{g}(P)\} = \varnothing]\!]^A\alpha_A(R)) = \\
[\![\mathbf{skip}]\!]^A([\![\mathtt{Is}(z)]\!]^A\alpha_A(R)) \sqcup_A [\![P_\omega]\!]^A\alpha(\varnothing) = \\
a \sqcup_A \bot_A = \\
a
\end{aligned}$$

and since $\bot_A \not\approx^{\varepsilon}_{\delta_A} a$ we can conclude that $f(P) \notin \mathbb{C}(\boldsymbol{A}, \varepsilon)$. Assume that A is incomplete on the check $[\![P]\!]\{\mathfrak{g}(P)\} \neq \varnothing$. This implies that the evaluation of the boolean guard of $f(P)$ is always true. By using the set of stores Q defined before, we get:

$$[\![f(P)]\!]^A\alpha_A(Q) = [\![\mathbf{skip}]\!]^A\alpha_A(Q) = \alpha_A(Q).$$

We know that Q is such that $\alpha_A(\varnothing) \not\approx^{\varepsilon}_{\delta_A} \alpha_A(Q)$, therefore $f(P) \notin \mathbb{C}(\boldsymbol{A}, \varepsilon)$. In both cases, we get an ε-partial incomplete program.

We can conclude that $P \in K \;\Leftrightarrow\; f(P) \in \mathbb{C}(\boldsymbol{A}, \varepsilon)$, thus $\overline{\mathbb{C}(\boldsymbol{A}, \varepsilon)}$ is productive. □

Corollary 4.28. *For every non-trivial $A \in \mathscr{A}(\wp^{\mathrm{re}}(\mathbb{S}))$, the class of incomplete programs $\overline{\mathbb{C}(A)}$ is productive.*

Proof. The proof coincides with the proof of Theorem 4.27 by replacing the predicate $a \not\approx^{\varepsilon}_{\delta_A} b$ with $a \neq b$. □

Let us notice that the proofs of Theorems 4.25 and 4.27 provide a further insight into the structure of $\mathbb{C}(\boldsymbol{A}, \varepsilon)$ and its complement class $\overline{\mathbb{C}(\boldsymbol{A}, \varepsilon)}$. These theorems prove that, given an abstract quasi-metric space of stores with unlimited imprecision $\boldsymbol{A}$, whenever we limit the expected imprecision of our analysis to a bound ε of possible false alarms, we cannot build a procedure that enumerates all programs satisfying that bound or that do not respect that bound, unless the abstract domain is trivial or with limited imprecision. Therefore, automating the proof that an abstract domain is partially complete or partially incomplete for a given program— i.e., a static program analysis can produce or cannot produce some bounded set of false alarms—is in general impossible. The partial completeness and incompleteness class of an abstraction are therefore a non-trivial property of programs for which no recursively enumerable procedure may exist which is able to enumerate all of their elements. Moreover, Theorem 4.27 proves that the partial incompleteness class $\overline{\mathbb{C}(\boldsymbol{A}, \varepsilon)}$ is productive. Recall that a set X is productive if there exists a general effective method (i.e., a total recursive function), also called the productivity function, which enables us to find, for every r.e. subset $Y \subseteq X$, an element $x \in X \smallsetminus Y$ [68]. Therefore, being $\overline{\mathbb{C}(\boldsymbol{A}, \varepsilon)}$ a productive set implies that any attempt to enumerate the set $\overline{\mathbb{C}(\boldsymbol{A}, \varepsilon)}$ with a program $P \in \mathsf{Prog}$ such that $W_P \subseteq \overline{\mathbb{C}(\boldsymbol{A}, \varepsilon)}$ (i.e., φ_P is a computable semi-characteristic function for $\overline{\mathbb{C}(\boldsymbol{A}, \varepsilon)}$), can be transformed, with an algorithmic program transformer $t : \mathsf{Prog} \to \mathsf{Prog}$, to another program $t(P) \in \mathsf{Prog}$ that is not in the enumeration W_P but in $\overline{\mathbb{C}(\boldsymbol{A}, \varepsilon)}$, namely, $t(P) \in \overline{\mathbb{C}(\boldsymbol{A}, \varepsilon)} \smallsetminus W_P$. We show the construction of such program transformer t in the following corollary.

Corollary 4.29. *Let $\boldsymbol{A} \in \mathfrak{A}(\wp^{\mathrm{re}}(\mathbb{S}))$ be some non-trivial abstract quasi-metric space with unlimited imprecision. Then, there exists a total recursive function (program transformer) $t : \mathsf{Prog} \to \mathsf{Prog}$ such that, for all $P \in \mathsf{Prog}$:*

$$W_P \subseteq \overline{\mathbb{C}(\boldsymbol{A}, \varepsilon)} \;\Rightarrow\; t(P) \in \overline{\mathbb{C}(\boldsymbol{A}, \varepsilon)} \smallsetminus W_P.$$

Proof. Theorem 4.27 proved that $\overline{\mathbb{C}(\boldsymbol{A}, \varepsilon)}$ is a productive set by showing the many-to-one reduction $\overline{K} \preceq_f \overline{\mathbb{C}(\boldsymbol{A}, \varepsilon)}$ by the total recursive function $f : \mathsf{Prog} \to \mathsf{Prog}$ defined in the proof of Theorem 4.27. Let $f^{-1}(X) \triangleq \{x \mid f(x) \in X\}$ for every set X. Clearly f^{-1} is monotone, thus

$$X \subseteq Y \Rightarrow f^{-1}(X) \subseteq f^{-1}(Y).$$

For all $P, Q \in \mathsf{Prog}$, we define the partial recursive function $\psi(P, Q)$ as follows

$$\psi(P,Q) \triangleq \begin{cases} 1 & \text{if } f(Q) \in W_P, \\ \uparrow & \text{otherwise.} \end{cases}$$

Since f is recursive, $\psi(P,Q)$ is partial recursive, therefore, by the s-m-n theorem, there exists a total recursive function $g : \mathsf{Prog} \to \mathsf{Prog}$ that specializes ψ with P, namely $\varphi_{g(P)}(Q) = \psi(P,Q)$ [1]. Note that, for all $P \in \mathsf{Prog}$, we have $W_{g(P)} = f^{-1}(W_P)$. Let us suppose that someone gives us a program $\hat{P}$ claiming that $\hat{P}$ enumerates all programs in $\overline{\mathbb{C}(\boldsymbol{A},\varepsilon)}$, i.e., $\hat{P}$ terminates on $Q \in \mathsf{Prog}$ if $Q \in \overline{\mathbb{C}(\boldsymbol{A},\varepsilon)}$. This means that $W_{\hat{P}} \subseteq \overline{\mathbb{C}(\boldsymbol{A},\varepsilon)}$. We show that

$$f(g(\hat{P})) \in \overline{\mathbb{C}(\boldsymbol{A},\varepsilon)} \setminus W_{\hat{P}}.$$

Because $W_{\hat{P}} \subseteq \overline{\mathbb{C}(\boldsymbol{A},\varepsilon)}$, by the monotonicity of f^{-1}, we have that $f^{-1}(W_{\hat{P}}) \subseteq f^{-1}(\overline{\mathbb{C}(\boldsymbol{A},\varepsilon)})$. By the reduction $\overline{K} \preceq_f \overline{\mathbb{C}(\boldsymbol{A},\varepsilon)}$, we know that $f^{-1}(\overline{\mathbb{C}(\boldsymbol{A},\varepsilon)}) = \overline{K}$ and also $f^{-1}(W_{\hat{P}}) = W_{g(\hat{P})}$. This means that $W_{g(\hat{P})} \subseteq \overline{K}$. We know that $\overline{K}$ is productive and the t function we are looking for in this corollary for $\overline{K}$ is the identity function. Therefore $g(\hat{P}) \in \overline{K} \setminus W_{g(\hat{P})}$ and by applying the reduction through f, we obtain $f(g(\hat{P})) \in \overline{\mathbb{C}(\boldsymbol{A},\varepsilon)}$. This means that $f(g(\hat{P}))$ is a partial incomplete program, therefore the proposed procedure $\hat{P}$, by assumption, must terminate on it, i.e., $f(g(\hat{P})) \in W_{\hat{P}}$. By the definition of function ψ, we have

$$\begin{aligned} f(g(\hat{P})) \in W_{\hat{P}} &\Rightarrow \psi(\hat{P}, g(\hat{P}))\downarrow \\ &\Leftrightarrow \varphi_{g(\hat{P})}(g(\hat{P}))\downarrow \\ &\Leftrightarrow g(\hat{P}) \in W_{g(\hat{P})} \\ &\Leftrightarrow g(\hat{P}) \in K \end{aligned}$$

but, by the productivity of $\overline{K}$, $g(\hat{P}) \in \overline{K} \setminus W_{g(\hat{P})}$, therefore we obtain a contradiction. This implies that $f(g(\hat{P}))$ can't be enumerated by $\hat{P}$, i.e., $f(g(\hat{P})) \in \overline{\mathbb{C}(\boldsymbol{A},\varepsilon)} \setminus W_{\hat{P}}$. By setting $t \triangleq \lambda P \in \mathsf{Prog}\,.\, f(g(P))$ we obtain the productivity function for $\overline{\mathbb{C}(\boldsymbol{A},\varepsilon)}$. □

The existence of a productivity function for the partial incompleteness class $\overline{\mathbb{C}(\boldsymbol{A},\varepsilon)}$ means that $\overline{\mathbb{C}(\boldsymbol{A},\varepsilon)}$ shares a structure that is similar to the set T of Gödel numbers of true sentences in an effective axiomatic system, like first-order arithmetic. Indeed, if W is a r.e. set of true sentences in first-order arithmetic, then, by the Gödel's first incompleteness theorem, there is at least one true sentence that is not in W.

[1] Here we assume that the specializer g does not add imprecision to any implementation in Prog of the partial recursive function ψ, i.e., g preserves completeness. This simply corresponds to syntactically substituting the input P in the program code of ψ without any code optimization.

We have seen that partial-completeness classes w.r.t. abstract quasi-metric spaces, inherit the non-recursive enumerability property from the known completeness class in abstract interpretation. A question now may arise: is there a *further weakening* that would allow us to be able to enumerate all programs in such new completeness class? That is, is there a "*tractable*", in terms of computability, completeness class such that the decidability problem is (at least) semi-decidable? The answer is yes, and, in the next section, we show a further requirement weakening on the partial completeness class that leads to r.e. classes of programs under a structural condition over the abstract domains of stores in $\mathscr{A}(\wp^{\mathrm{re}}(\mathbb{S}))$.

4.4 Locally Partial Complete Abstractions of Stores

The weaker notion of *local completeness* has been recently introduced by Bruni et al. in [9] that requires completeness only with respect to specific inputs. This new class of programs is formally defined as:

$$\mathbb{C}(A, S) \triangleq \{P \in \mathsf{Prog} \mid \alpha_A(\llbracket P \rrbracket S) = \llbracket P \rrbracket^A \alpha_A(S)\}.$$

It is clear that the *global* notion of completeness in [27,43] implies local completeness, while the converse does not hold in general. This because local completeness is relative to a specific input $S \in \wp^{\mathrm{re}}(\mathbb{S})$, while completeness is relative to the whole concrete domain $\wp^{\mathrm{re}}(\mathbb{S})$. We follow a parallel approach and, by requiring partial completeness on a fixed set of inputs only, we obtain a further weakening on the definition of ε-partial completeness defined in the previous sections and of the notion of local completeness.

Definition 4.30 (Local ε-partial completeness). *Consider a program $P \in \mathsf{Prog}$, a non-empty set of input stores $S \in \wp^{\mathrm{re}}(\mathbb{S})$ and a constant bound $\varepsilon \in \mathbb{Q}_{\geq 0}$. An abstract quasi-metric space of stores $\boldsymbol{A} \in \mathfrak{A}(\wp^{\mathrm{re}}(\mathbb{S}))$ is locally ε-partial complete for P in S if:*

$$\alpha_A(\llbracket P \rrbracket S) \approx^{\varepsilon}_{\delta_A} \llbracket P \rrbracket^A \alpha_A(S).$$

We introduce the notion of *local ε-partial completeness* which in program analysis corresponds to consider ε-partial completeness only along certain program traces, i.e., on a (r.e.) set of inputs.

Definition 4.31 (Local ε-partial completeness class). *Given an abstract quasi-metric space $\boldsymbol{A} \in \mathfrak{A}(\wp^{\mathrm{re}}(\mathbb{S}))$, a constant $\varepsilon \in \mathbb{Q}_{\geq 0}$ and a non-empty set of inputs $\mathfrak{F} \subseteq \wp^{\mathrm{re}}(\mathbb{S})$, we define $\mathbb{C}(\boldsymbol{A}, \mathfrak{F}, \varepsilon) \subseteq \mathsf{Prog}$ as the class of programs which are ε-partially complete for $\boldsymbol{A}$ only along the set of inputs $\mathfrak{F}$. Formally:*

$$\mathbb{C}(\boldsymbol{A}, \mathfrak{F}, \varepsilon) \triangleq \{P \in \mathsf{Prog} \mid \forall S \in \mathfrak{F}.\ \alpha_A(\llbracket P \rrbracket S) \approx^{\varepsilon}_{\delta_A} \llbracket P \rrbracket^A \alpha_A(S)\}.$$

Because recursive and trivial abstract domains of stores are either finite or countably infinite, we can define $\mathbb{C}(\boldsymbol{A}, \mathfrak{F}, \infty) \triangleq \cup_{\varepsilon \in \mathbb{Q}_{\geq 0}} \mathbb{C}(\boldsymbol{A}, \mathfrak{F}, \varepsilon)$. The following proposition is a straightforward consequence of Definition 4.31 of local partial-completeness.

Proposition 4.32. *For all* $\boldsymbol{A} \in \mathfrak{A}(\wp^{\mathrm{re}}(\mathbb{S}))$, $S \in \wp^{\mathrm{re}}(\mathbb{S})$ *and* $\varepsilon \in \mathbb{Q}_{\geq 0}$ *the following holds:*

(i) $\mathbb{C}(A) \subseteq \mathbb{C}(A, S) = \mathbb{C}(\boldsymbol{A}, S, 0) \subseteq \mathbb{C}(\boldsymbol{A}, S, \varepsilon)$

(ii) $\forall \varepsilon, \xi \in \mathbb{Q}_{\geq 0}.\ \varepsilon \leq \xi \ \Rightarrow\ \mathbb{C}(\boldsymbol{A}, S, \varepsilon) \subseteq \mathbb{C}(\boldsymbol{A}, S, \xi)$;

(iii) $\forall \mathfrak{F}, \mathfrak{F}' \subseteq \wp^{\mathrm{re}}(\mathbb{S}).\ \mathfrak{F} \subseteq \mathfrak{F}' \ \Rightarrow\ \mathbb{C}(\boldsymbol{A}, \mathfrak{F}', \varepsilon) \subseteq \mathbb{C}(\boldsymbol{A}, \mathfrak{F}, \varepsilon)$

(iv) $\forall \mathfrak{F}, \mathfrak{F}' \subseteq \wp^{\mathrm{re}}(\mathbb{S}).\ \mathbb{C}(\boldsymbol{A}, \mathfrak{F} \cup \mathfrak{F}', \varepsilon) = \mathbb{C}(\boldsymbol{A}, \mathfrak{F}, \varepsilon) \cap \mathbb{C}(\boldsymbol{A}, \mathfrak{F}', \varepsilon)$

(v) $\mathbb{C}(\boldsymbol{A}, S, \infty) = \mathsf{Prog}$.

Forcing a zero closeness means requiring no false alarms on input S, i.e., local completeness, while each complete abstraction is also locally partial complete for every S. Moreover, weakening closeness increases monotonically the set of locally partial complete programs, while by increasing the considered set of inputs we might restrict the local partial completeness class. From now on, we will focus on classes of local partial completeness with a single input that will be denoted with S instead of $\{S\}$.

Example 4.33. Consider the abstract quasi-metric space $(\mathsf{Int}, \delta^{\mathfrak{w}}_{\mathsf{Int}}) \in \mathfrak{A}(\wp^{\mathrm{re}}(\mathbb{Z}))$. Let us consider the class of 0-partial complete programs w.r.t. $(\mathsf{Int}, \delta^{\mathfrak{w}}_{\mathsf{Int}})$, i.e., $\mathbb{C}((\mathsf{Int}, \delta^{\mathfrak{w}}_{\mathsf{Int}}), S, 0)$ which does not admit any imprecision on $S \in \wp^{\mathrm{re}}(\mathbb{S})$, and the class $\mathbb{C}((\mathsf{Int}, \delta^{\mathfrak{w}}_{\mathsf{Int}}), S, 1)$, which admits only one spurious element on the interval output. We define $P_{\mathrm{abs}} \in \mathsf{Prog}$ as:

$$\begin{aligned} \mathrm{P}_{\mathrm{abs}} \ \triangleq\ & \textbf{if } x \geq 0 \textbf{ then skip} \\ & \textbf{else } x := x * (-1) \end{aligned}$$

which, computes the absolute value of integer variables. Consider the input sets $S_1 \triangleq \{0, 2, 5\}$ and $S_2 \triangleq \{-1, 3, 7\}$. We have the following concrete and abstract evaluations:

$$\begin{aligned} \alpha_{\mathsf{Int}}([\![P_{\mathrm{abs}}]\!]S_1) = \alpha_{\mathsf{Int}}([\![\textbf{skip}]\!]S_1) = \alpha_{\mathsf{Int}}(S_1) = [0, 5] \\ [\![P_{\mathrm{abs}}]\!]^{\mathsf{Int}}\alpha_{\mathsf{Int}}(S_1) = [\![P_{\mathrm{abs}}]\!]^{\mathsf{Int}}[0, 5] = [\![\textbf{skip}]\!]^{\mathsf{Int}}[0, 5] = [0, 5] \end{aligned}$$

$$\begin{aligned}
\alpha_{\mathsf{Int}}([\![P_{\text{abs}}]\!]S_2) &= \alpha_{\mathsf{Int}}([\![\textbf{skip}]\!]\{3,7\} \cup [\![x := x * (-1)]\!]\{-1\}) \\
&= \alpha_{\mathsf{Int}}(\{1,3,7\}) \\
&= [1,7] \\
[\![P_{\text{abs}}]\!]^{\mathsf{Int}}\alpha_{\mathsf{Int}}(S_2) &= [\![\textbf{skip}]\!]^{\mathsf{Int}}[0,7] \sqcup_{\mathsf{Int}} [\![x := x * (-1)]\!]^{\mathsf{Int}}[-1,-1] \\
&= [0,7] \sqcup_{\mathsf{Int}} [1,1] \\
&= [0,7]
\end{aligned}$$

Then, clearly $P_{\text{abs}} \in \mathbb{C}((\mathsf{Int}, \delta^{\mathfrak{w}}_{\mathsf{Int}}), S_1, 0)$, $P_{\text{abs}} \notin \mathbb{C}((\mathsf{Int}, \delta^{\mathfrak{w}}_{\mathsf{Int}}), S_2, 0)$, while $P_{\text{abs}} \in \mathbb{C}((\mathsf{Int}, \delta^{\mathfrak{w}}_{\mathsf{Int}}), S_2, 1)$. ■

The following theorem shows that the class of programs $\mathbb{C}(\boldsymbol{A}, S, \varepsilon)$ turns out to be r.e. if the abstract domain of stores A meets the ACC property.

Theorem 4.34. *If $\boldsymbol{A} \in \mathfrak{A}(\wp^{\text{re}}(\mathbb{S}))$ is ACC, then for all $S \in \wp^{\text{re}}(\mathbb{S})$ and $\varepsilon \in \mathbb{Q}_{\geq 0}$, $\mathbb{C}(\boldsymbol{A}, S, \varepsilon)$ is recursively enumerable.*

Proof. Clearly, if $A = \top^{\mathbb{S}}$ then $\mathbb{C}(\boldsymbol{A}, S, \varepsilon) = \mathsf{Prog}$ and therefore r.e. (recursive in this case). Suppose $A \neq \top^{\mathbb{S}}$ and ACC. We write the pseudo-code of a computable procedure that is able to check whether $P \in \mathsf{Prog}$ is local ε-partial complete for $\boldsymbol{A}$ with input S, while it does not terminate on inputs $P \notin \mathbb{C}(\boldsymbol{A}, S, \varepsilon)$. Therefore, without loss of generality, let us define ψ as the partial function

$$\psi : \mathsf{Prog} \times \mathfrak{A}^{\text{ACC}}(\wp^{\text{re}}(\mathbb{S})) \times \wp^{\text{re}}(\mathbb{S}) \times \mathbb{Q}_{\geq 0} \nrightarrow \mathsf{Prog}$$

defined as:

$$\psi(P, \boldsymbol{A}, S, \varepsilon) \triangleq \begin{cases} P & \text{if } P \in \mathbb{C}(\boldsymbol{A}, S, \varepsilon), \\ \uparrow & \text{otherwise} \end{cases}$$

where we use the notation $\mathfrak{A}^{\text{ACC}}(\wp^{\text{re}}(\mathbb{S}))$ to denote the set of all abstract quasi-metric spaces with A satisfying the ACC. The high level steps of a possible algorithm $\mathfrak{P} \in \mathsf{Prog}$ that implements ψ are the following:

(1) run the abstract interpreter $[\![P]\!]^A$ with input the recursive set of stores $\alpha_A(S)$. Suppose $[\![P]\!]^A\alpha_A(S) = a$;

(2) if $a = \bot_A$, then return P (i.e., P is locally complete by the soundness property of abstract interpretation);

(3) otherwise ($a \neq \bot_A$), let us consider an enumeration $\{s_i\}_{i\in\mathbb{N}}$ of all $s \in S$ (recall that S is r.e. by Assumption 1);

(4) let $b \triangleq \bot_A$. Run a dovetail algorithm constructing the set

$$b \triangleq \bigsqcup_{i\in\mathbb{N}} \{\alpha_A([\![P]\!]s_i)\}.$$

The dovetail algorithm performs the first step of program $[\![P]\!]s_0$ on the first store $s_0 \in S$; next, perform the first step on the second store s_1 and the second step on the first store s_0; next, perform the first step of the third store s_2, the second step of the second store s_1, and the third step on the first store s_0; and so on. When the program $[\![P]\!]s_i$ terminates (i.e., $[\![P]\!]s_i \neq \varnothing$) on some $s_i \in S$, then (since α_A is total recursive) the element $\alpha_A([\![P]\!]s_i)$ is added to b, namely:

$$b \triangleq b \sqcup_A \alpha_A([\![P]\!]s_i)$$

(5) if, at some point, there exists $s_i \in S$ such that $[\![P]\!]s_i \neq \varnothing$, $b = b \sqcup_A \alpha_A([\![P]\!]s_i)$ and $\delta_A(b, a) \leq \varepsilon$ (recall that $\delta_A(b, a) \leq \varepsilon$ is a decidable predicate by Definition 4.1 of quasi-metric A-compatible) then return P.

If $P \in \mathbb{C}(\boldsymbol{A}, S, \varepsilon)$ then the convergence of the equation $b = b \sqcup_A \alpha_A([\![P]\!]s_i)$ is guaranteed by the ACC property assumption of $\boldsymbol{A}$. It is easy to note that the procedure defined above is algorithmic and therefore program $\mathfrak{P}$ computes the partial function ψ. Let $\varphi_{\mathfrak{P}}$ be the partial recursive function associated to program $\mathfrak{P}$ (and having index $\mathfrak{P}$). Clearly, we have $\varphi_{\mathfrak{P}} \cong \psi$. By the s-m-n theorem, we can specialize $\varphi_{\mathfrak{P}}$ with inputs $\boldsymbol{A}$, S and ε. That is, there exists a total recursive function $s : \mathsf{Prog} \times \mathfrak{A}^{\mathrm{ACC}}(\wp^{\mathrm{re}}(\mathbb{S})) \times \wp^{\mathrm{re}}(\mathbb{S}) \times \mathbb{Q}_{\geq 0} \to \mathsf{Prog}$ such that for all $P \in \mathsf{Prog}$: $\varphi_{s(\mathfrak{P},\boldsymbol{A},S,\varepsilon)}(P) = \varphi_{\mathfrak{P}}(P, \boldsymbol{A}, S, \varepsilon)$. Moreover, we have:

$$range(\varphi_{s(\mathfrak{P},\boldsymbol{A},S,\varepsilon)}(P)) = \mathbb{C}(\boldsymbol{A}, S, \varepsilon).$$

This proves that $\mathbb{C}(\boldsymbol{A}, S, \varepsilon)$ is recursively enumerable. □

The general algorithm proposed in the proof of Theorem 4.34, allows us to systematically prove the local ε-partial completeness of any program $P \in \mathsf{Prog}$ on a single input $S \in \wp^{\mathrm{re}}(\mathbb{S})$ w.r.t. any ACC abstract quasi-metric space $\boldsymbol{A}$, while it does not give an answer to programs $P \notin \mathbb{C}(\boldsymbol{A}, S, \varepsilon)$.

Example 4.35. Consider the $(\mathsf{Sign}, \delta^{\mathfrak{w}}_{\mathsf{Sign}}) \in \mathfrak{A}(\wp^{\mathrm{re}}(\mathbb{Z}))$ abstract quasi-metric space and the following program

$$P \triangleq \textbf{if } x \geq 1 \textbf{ then } x := x - 1 \textbf{ else skip}$$

We want to check whether P is in $\mathbb{C}((\mathsf{Sign}, \delta^{\mathfrak{w}}_{\mathsf{Sign}}), \mathbb{Z}_{\geq 0}, 1)$, i.e., the class of programs whose concrete and abstract evaluations on the positive integers are 1-close. Following the algorithm $\mathfrak{P}$ above, from the abstract interpreter we get:

$$[\![P]\!]^{\mathsf{Sign}} \alpha_{\mathsf{Sign}}(\mathbb{Z}_{\geq 0}) = \mathbb{Z}.$$

By constructing the set $b = b \sqcup_{\mathsf{Sign}} \alpha_{\mathsf{Sign}}([\![P]\!]s_i)$ where $s_i \in \mathbb{Z}_{\geq 0}$ is an enumeration of $\mathbb{Z}_{\geq 0}$, we get:

$$
\begin{array}{rcl}
b & = & \varnothing \\
b & = & \varnothing \sqcup_{\mathsf{Sign}} \alpha_{\mathsf{Sign}}([\![P]\!]\{0\}) \;=\; 0 \;\not\approx^{1}_{\delta^{\mathfrak{w}}_{\mathsf{Sign}}} \mathbb{Z} \\
b & = & 0 \sqcup_{\mathsf{Sign}} \alpha_{\mathsf{Sign}}([\![P]\!]\{1\}) \;=\; 0 \sqcup_{\mathsf{Sign}} 0 \;=\; 0 \;\not\approx^{1}_{\delta^{\mathfrak{w}}_{\mathsf{Sign}}} \mathbb{Z} \\
b & = & 0 \sqcup_{\mathsf{Sign}} \alpha_{\mathsf{Sign}}([\![P]\!]\{2\}) \;=\; 0 \sqcup_{\mathsf{Sign}} + \;=\; + \;\approx^{1}_{\delta^{\mathfrak{w}}_{\mathsf{Sign}}} \mathbb{Z}
\end{array}
$$

at this point the algorithm terminates, hence we can conclude that P is locally 1-partial complete for $(\mathsf{Sign}, \delta^{\mathfrak{w}}_{\mathsf{Sign}})$, namely, $P \in \mathbb{C}((\mathsf{Sign}, \delta^{\mathfrak{w}}_{\mathsf{Sign}}), \mathbb{Z}_{\geq 0}, 1)$. Note that, if we check whether P is in $\mathbb{C}((\mathsf{Sign}, \delta^{\mathfrak{w}}_{\mathsf{Sign}}), \mathbb{Z}_{\geq 0}, 0)$, then the algorithm keeps running the check

$$
b \;=\; + \sqcup_{\mathsf{Sign}} \alpha_{\mathsf{Sign}}([\![P]\!]\{n\}) \;=\; + \sqcup_{\mathsf{Sign}} + \;=\; + \;\not\approx^{0}_{\delta^{\mathfrak{w}}_{\mathsf{Sign}}} \mathbb{Z}
$$

for all $n \in \mathbb{Z}_{\geq 0}$. This because $P \notin \mathbb{C}((\mathsf{Sign}, \delta^{\mathfrak{w}}_{\mathsf{Sign}}), \mathbb{Z}_{\geq 0}, 0)$, indeed $P \notin \mathbb{C}(\mathsf{Sign})$. ■

We can trivially extend the use of the algorithm proposed in Theorem 4.34 to the class of programs which are complete with respect to ACC abstract domains of stores and a set of input stores, i.e., the local completeness class.

Corollary 4.36. *For all $A \in \mathscr{A}(\wp^{\mathrm{re}}(\mathbb{S}))$ and $S \in \wp^{\mathrm{re}}(\mathbb{S})$, if A is ACC, then the class of programs $\mathbb{C}(A, S)$ is recursively enumerable.*

Proof. The proof follows from the proof of Theorem 4.34 by substituting the check $\delta(b, a) \leq \varepsilon$ with $b = a$. □

The following theorem proves that both $\mathbb{C}(\boldsymbol{A}, S, \varepsilon)$ and $\overline{\mathbb{C}(\boldsymbol{A}, S, \varepsilon)}$ are non-r.e. sets when $\boldsymbol{A}$ is not ε-trivial and not ACC.

Theorem 4.37. *If $\boldsymbol{A} \in \mathfrak{A}(\wp^{\mathrm{re}}(\mathbb{S}))$ is not ε-trivial, then $\overline{\mathbb{C}(\boldsymbol{A}, S, \varepsilon)}$ is non-r.e.. Moreover, if $\boldsymbol{A}$ is also not ACC, then $\mathbb{C}(\boldsymbol{A}, S, \varepsilon)$ is non-r.e..*

Proof. The proof follows immediately by Theorem 4.25, as the class can be characterized by the first order predicate $R(\boldsymbol{A}, \varepsilon, P, s, n)$ where:

$$
P \in \overline{\mathbb{C}(\boldsymbol{A}, S, \varepsilon)} \;\Leftrightarrow\; \exists s \in S.\, \forall n \in \mathbb{N}.\, \neg R(\boldsymbol{A}, \varepsilon, P, s, n) \in \Sigma_2.
$$

□

Corollary 4.38. *If $A \notin \{id, \top^{\mathbb{S}}\}$ then the local incompleteness class $\overline{\mathbb{C}(A, S)}$ is non-r.e..*

We conclude by showing in Table 4.1 all the recursive properties of each completeness, partial, local and local partial completeness classes presented in this chapter. Let us stress the reader about the three important assumptions we made that are the foundation for the recursivity results presented in this chapter and summarized in Table 4.1:

	Class	Recursive Property	Proof	Conditions
Completeness	$\mathbb{C}(A)$	non-r.e.	Corollary 4.26	A non-trivial
	$\overline{\mathbb{C}(A)}$	productive	Corollary 4.28	A non-trivial
Partial Completeness	$\mathbb{C}(\boldsymbol{A}, \varepsilon)$	non-r.e.	Theorem 4.25	$\boldsymbol{A}$ non-ε-trivial
	$\overline{\mathbb{C}(\boldsymbol{A}, \varepsilon)}$	productive	Theorem 4.27	$\boldsymbol{A}$ non-trivial, unlimited error
Local Completeness	$\mathbb{C}(A, S)$	r.e.	Corollary 4.36	A ACC
	$\overline{\mathbb{C}(A, S)}$	non-r.e.	Corollary 4.38	A non-trivial
Local Partial Completeness	$\mathbb{C}(\boldsymbol{A}, S, \varepsilon)$	r.e.	Theorem 4.34	$\boldsymbol{A}$ ACC
	$\overline{\mathbb{C}(\boldsymbol{A}, S, \varepsilon)}$	non-r.e.	Theorem 4.37	$\boldsymbol{A}$ non-trivial, unlimited error

Table 4.1: Recursive properties of completeness, partial completeness, local completeness and local partial completeness classes of programs and their respective complements classes

(i) the non-trivial abstract domains of stores are recursive abstractions (Assumption 4), i.e., abstract domains $A \in \mathscr{A}(\wp^{\mathrm{re}}(\mathbb{S}))$ for which it is decidable to check whether a concrete (finite) state $s \in \mathbb{S}$ holds an abstract property $a \in A$. This hypothesis is satisfied in all frameworks for static program analysis of course. Indeed, weakening recursivity of the abstract domain would mean that it may be impossible to check whether the computed store satisfies a property at any given program point, making it impossible to check alarms;

(ii) we considered a fixed, inductively defined, concrete and abstract denotational semantics of while-programs defined respectively in Figure 2.2 and 3.5. This means that all the completeness classes considered in this chapter do not quantify over all the possible concrete and abstract program semantics;

(iii) the abstract interpreter is specified by a GI (Assumption 2) and every non-trivial abstract domain of stores in $\mathscr{A}(\wp^{\mathrm{re}}(\mathbb{S}))$ is strict (Assumption 4). As a consequence of this assumption and the previous point (ii), every non-trivial abstract domain of stores both exactly represents the empty set, i.e., $\gamma_A(\alpha_A(\varnothing)) = \varnothing$, and it is complete respect to the non-terminating program $P_w \triangleq$ **while t do skip**. This observation can be formalized as

follows:

$$\forall \boldsymbol{A} \in \mathfrak{A}(\wp^{\mathrm{re}}(\mathbb{S})).\ A \notin \{id, \top^{\mathbb{S}}\} \ \Rightarrow\ P_w \in \mathbb{C}(A) \subseteq \mathbb{C}(\boldsymbol{A}, \varepsilon) \subseteq \mathbb{C}(\boldsymbol{A}, S, \varepsilon).$$

Although the non-recursive enumerability property establishes that automating the proof that an abstraction is ε-partial complete or incomplete for a given program, is impossible, this does not exclude the possibility to provide r.e. under-approximations of $\mathbb{C}(\boldsymbol{A}, \varepsilon)$ and $\overline{\mathbb{C}(\boldsymbol{A}, \varepsilon)}$, as done in [44] for the class $\mathbb{C}(A)$.

Chapter 5

ABSTRACT INTERPRETATION OF INDEXED GRAMMARS

The scope of the abstract interpretation framework is not limited only to static program analysis. Furthermore, the (in)completeness property of an abstract function on a generic abstract domain may encode different meanings besides the optimal precision in static program analysis. In this section, we aim to show that Chomsky's hierarchy of formal languages can be understood by abstract interpretation between grammars. Moreover, known separation results between classes of formal languages can be generalized as instances of *incompleteness* of grammars abstractions. In particular, we demonstrate one such step between indexed grammars and CF grammars. *Indexed grammars* are a generalization of CF grammars and recognize a proper subset of CS languages. We formally define the structure of indexed grammars in Section 5.1 and their recognized languages which form the family of *indexed languages*. We then characterize in Section 5.2 indexed languages with a *fixpoint* of an equation system over "*indexed states*" (a vector of suitable languages). The equation system is constructed in the form of π-functions consisting of a substitution function and a regular language. Building on the equational fixpoint characterization, in Section 5.3 we present three abstractions of indexed grammars. These come in the form of closure operators over the fixpoint's underlying indexed states: stack elimination (Section 5.3.1), stack bounding by k-limiting (Section 5.3.2), and stack-copy limiting (Section 5.3.3). Overall the three abstractions are proven sound and are characterized by approximate fixpoints. Finally, in Section 5.4 we show that the indexed languages not expressible through a CF grammar, correspond to *witnesses of the incompleteness* of stack elimination abstraction, thus providing a new point of view of the (in)completeness class for the grammars abstractions. This chapter is mainly based on [11].

5.1 Indexed Languages

Indexed grammars were introduced by Aho in the late 1960s to model a natural subclass of context-sensitive languages, more expressive than context-free grammars with interesting closure properties [2]. In the following, we use the definition of indexed grammar provided in [1].

Definition 5.1. *An Indexed Grammar is a* 5*-tuple* $G \triangleq (N, T, I, P, S)$ *such that:*

(1) N*,* T *and* I *are three mutually disjoint finite sets of symbols: the set* N *of non-terminals, the set* T *of terminals and the set* I *of indices, where* ϵ *is a designated symbol for the empty sequence;*

(2) $S \in N$ *is a distinguished symbol in* N*, namely the start symbol;*

(3) P *is a finite set of productions, each having the form of one of the following:*

(a) $A \to \alpha$ *(Stack copy)*

(b) $A \to B_f$ *(Push)*

(c) $A_f \to \beta$ *(Pop)*

where $A, B \in N$ *are non-terminal symbols,* $f \in I$ *is an index symbol and* $\alpha, \beta \in (N \cup T)^*$*.*

Observe the similarity to context-free grammars which are only defined by production rules of type (*3a*). The above definition is a finite representation of rules that rewrite pairs of non-terminal and sequences of index symbols that we call stacks. A key feature of indexed grammars is that their productions in P expand non-terminal/stack pairs of the form (A, σ), where $A \in N$ and $\sigma \in I^*$. So each non-terminal symbol $A \in N$ together with its stack $\sigma \in I^*$, can be viewed as a pair (A, σ) and the start symbol S is shorthand for the pair (S, ϵ). Therefore, with a slight abuse of notation, in the rest of this chapter we use α, β, γ to denote strings of terminal symbols and non-terminals symbols with its stack, namely, $\alpha, \beta, \gamma \in ((N \times I^*) \cup T)^*$. A string $\beta \in ((N \times I^*) \cup T)^*$ is often referred to as a *sentential form*. Given some non-empty stack $\sigma \in I^+$, the top symbol is the left-most index. The stack is implicit and is copied, to all non-terminals only, when the production is applied. So, for example, the type (*3a*) production rule $A \to aBC$ is a shorthand for $(A, \sigma) \to a(B, \sigma)(C, \sigma)$ with $A, B, C \in N$, $a \in T$ and $\sigma \in I^*$. A production rule of the form (*3b*) implements a push onto the stack while a production rule of the form (*3c*) encodes a pop off of the stack. For example, the production rule $A \to B_f$ applied to (A, σ) expands to $(B, f\sigma)$ where $f\sigma$ is the stack with the index $f \in I$ pushed on. Likewise, $A_f \to \beta$ can only be applied to (A, σ) if the top of the stack string σ is f. The result is β such that any non-terminal $B \in \beta$ is of the form (B, σ'), where σ' is the stack with

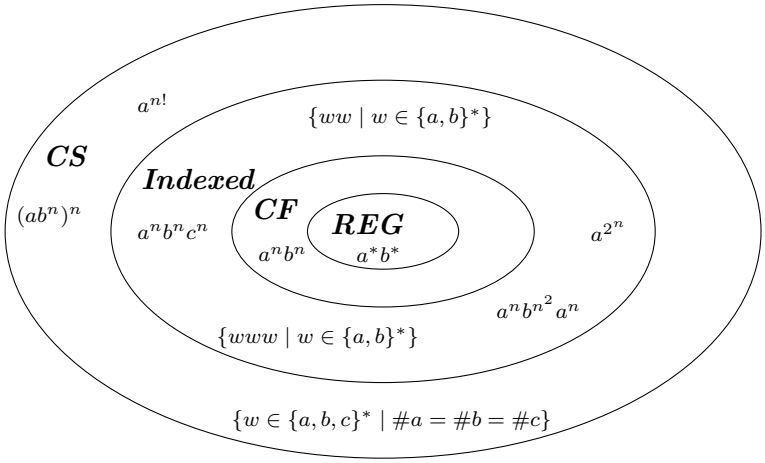


Figure 5.1: Chomsky hierarchy

the top character f popped off. We will use the symbols f, g to denote index symbols in I, i.e., $f, g \in I$. Push and Pop productions differ from the original definition given by Aho [2] in which, by Definition 5.1, at most one index symbol is loaded or unloaded in any production.

The language $\mathscr{L}(G)$ recognized by an indexed grammar G is

$$\mathscr{L}(G) \triangleq \{w \in T^* \mid (S, \epsilon) \rightarrow_G^* w\}.$$

We denote with $IL \subset \wp(T^*)$ the set of *indexed languages*.

Example 5.2. The language $L \triangleq \{a^n b^n c^n \mid n \geq 1\}$ is generated by the indexed grammar $G \triangleq (\{S, T, A, B, C\}, \{a, b, c\}, \{f\}, P, S)$, with productions in P:

$$\begin{array}{lll} S \rightarrow T & A_f \rightarrow aA & B_\epsilon \rightarrow b \\ T \rightarrow T_f & A_\epsilon \rightarrow a & C_f \rightarrow cC \\ T \rightarrow ABC & B_f \rightarrow bB & C_\epsilon \rightarrow c \end{array}$$

For example, the word "*aabbcc*" can be generated by the following derivations:

$$\begin{aligned} (S, \epsilon) &\rightarrow (T, \epsilon) \\ &\rightarrow (T, f) \\ &\rightarrow (A, f)(B, f)(C, f) \\ &\rightarrow a(A, \epsilon)(B, f)(C, f) \\ &\rightarrow aa(B, f)(C, f) \\ &\rightarrow^* aabbcc. \end{aligned}$$

■

Class	Emptiness	Membership	Equivalence
Regular	$\mathbb{P}\ (O(n))$	$\mathbb{P}\ (O(n))$	NL-complete
Context-free	$\mathbb{P}\ (O(n^3))$	$\mathbb{P}\ (O(n^3))$	Undecidable
Indexed	EXP-complete	NP-complete	Undecidable
Context-sensitive	Undecidable	PSPACE-complete	Undecidable

Table 5.1: Decidability and complexity results of known classes of formal languages

Indexed languages are recognized by *nested stack automata* [3]. A nested stack automaton is a finite automaton that can make use of a stack containing data which can be additional stacks. Like a stack automaton, a nested stack automaton may step up or down in the stack, and read the current symbol; in addition, it may at any place create a new stack, operate on that one, eventually destroy it, and continue operating on the old stack. In this way, stacks can be nested recursively to an arbitrary depth; however, the automaton always operates on the innermost stack only. For more details on nested stack automata see [3]. As argued above, the class of indexed languages properly includes the one of CF languages, while being properly included in the one of CS languages. Figure 5.1 represents these different classes and highlights some of the languages that characterize the different classes [64]. Table 5.1 reports some decidability and computational complexity properties of indexed languages and of some of the most known formal languages in the Chomsky hierarchy, where $\mathbb{P}$ denotes the polynomial time complexity class. As expected, the decidability results of indexed languages lay in between the ones of CS and CF languages.

Another characterization of indexed languages was given by Maslov [58] as the set of strings accepted by order-2 pushdown automata which are part of the higher-order pushdown automata or, equivalently, by higher-order grammars, i.e., extension of context-free grammars where non-terminals are functions taking parameters as inputs. Higher-order grammars, such as indexed grammars, are natural models used for analysis and verification of higher-order programs, e.g., functional [52,53,63] and logic [6] programs. We can also mention their use in natural language analysis [41].

5.2 Fixpoint Characterization of Indexed Languages

In order to study the existence of abstraction functions between context-sensitive, indexed and context-free languages, we need to provide a *fixpoint*

characterization of indexed languages. The fixpoint characterizations of CF languages, well known as the ALGOL-like theorem, and CS languages are already constructed and proved in [47, 50].

The fixpoint characterization that we present is mainly derived from the one of CS languages [50]. Essentially it consists of two elements: a *substitution function* that simulates a context-free rule for the pair non-terminal/stack (productions of type (*3a*) and (*3b*)) and a *regular expression* to verify the context of the stack in case of a pop production (*3c*). Before showing the theorem, let us give some notations and definitions. We denote with $\boldsymbol{X} \triangleq (X_1, \ldots, X_n)$ a tuple $\boldsymbol{X}$ of $n \in \mathbb{N}_{>0}$ elements. Let V be the set of variables of an indexed grammar: an element of V is either a pair of non-terminal/stack or a terminal symbol, namely, $V \triangleq (N \times I^*) \cup T$. Therefore, if β is a sentential form of an indexed grammar, then $\beta \in V^*$.

Definition 5.3 (Indexed state). *An indexed state is a n-tuple $\boldsymbol{X} \triangleq (X_1, \ldots, X_n)$ of sets of sentential forms $X_i \in \wp(V^*)$ with $i \in [1, n]$.*

Thus, the set of possible n-tuples of indexed states is $\underbrace{\wp(V^*) \times \cdots \times \wp(V^*)}_{n}$.

Definition 5.4 (Substitution function). *A substitution function h is a map of sets of sentential forms: $h : \wp(V^*) \to \wp(V^*)$.*

Functions h will be defined in the proof of Theorem 5.7 and will be used to simulate an application of a CF-like production.

Regular sets over V are denoted by regular expressions $R \in \wp(V^*)$ and will be used to verify the top character of the stack for pop productions.

Definition 5.5 (π-function). *Given a substitution function h and a regular set R, we define a pair $\pi \triangleq (h, R)$ called a π-function. A π-function is a map*

$$\pi : \underbrace{\wp(V^*) \times \cdots \times \wp(V^*)}_{n} \to \wp(V^*)$$

defined as follows:

$$\pi(\boldsymbol{X}) \triangleq h(\bigcup \boldsymbol{X} \cap R)$$

where $\bigcup \boldsymbol{X} \triangleq X_1 \cup X_2 \cup \cdots \cup X_n$ is the set corresponding to the union of all components of the indexed state $\boldsymbol{X}$.

Definition 5.6 (Vector of π-functions). *We denote with the bold symbol*

$$\boldsymbol{\pi} : \underbrace{\wp(V^*) \times \cdots \times \wp(V^*)}_{n} \to \underbrace{\wp(V^*) \times \cdots \times \wp(V^*)}_{n}$$

a vector function of π-functions $\boldsymbol{\pi} \triangleq (\pi_1, \ldots, \pi_n)$ such that

$$\boldsymbol{\pi}(\boldsymbol{X}) \triangleq (\pi_1(\boldsymbol{X}), \ldots, \pi_n(\boldsymbol{X})).$$

As we will see in the proof of Theorem 5.7, a π-function allows us to simulate an application of an indexed production, while a vector of π-functions $\boldsymbol{\pi}$ collects all the π-functions associated to each production in P plus two more π-functions to handle terminal words and sentential forms.

We can now prove the following:

Theorem 5.7. *Let G be an indexed grammar with m productions in P. Then $\mathscr{L}(G)$ is a component of the least fixpoint of a system of equations on indexed states:*

$$\boldsymbol{X}_G^{j+1} = \boldsymbol{\pi}_G(\boldsymbol{X}_G^j) \tag{5.1}$$

where $\boldsymbol{X}_G$ has $n \triangleq m + 2$ components and

$$\boldsymbol{X}_G^0 \triangleq \underbrace{(\{(S, \epsilon)\}, \varnothing, \ldots, \varnothing)}_{n}$$

is the initial indexed state, while the vector function of π-functions induced by G is

$$\boldsymbol{\pi}_G \triangleq (\pi_{G,1}, \ldots, \pi_{G,n}).$$

Proof. The proof considers an indexed grammar $G \triangleq (N, T, I, P, S)$ with m productions in P. Let $[\,\tau\,]_i \in P$ be an enumeration of all productions in P with $i \in [1, m]$, where $\tau \in P$ could be in one of the three forms of Definition 5.1. We build a system of equations having the form (5.1) where each indexed state $\boldsymbol{X}_G$ has $n \triangleq m + 2$ components: one for each production in P, namely, $X_{G,1}, \ldots, X_{G,m}$, and two additional components $X_{G,0}$ and $X_{G,t}$, where t is a variable symbol denoting the "terminals" set and it is always at position $m + 1$ of $\boldsymbol{X}_G$. So the components of each indexed state are $\boldsymbol{X}_G \triangleq (X_{G,0}, X_{G,1}, \ldots, X_{G,m}, X_{G,t})$. The least fixpoint computation of the so obtained system of equations is calculated by the iterative application of the vector function $\boldsymbol{\pi}_G \triangleq (\pi_{G,0}, \pi_{G,1}, \ldots, \pi_{G,m}, \pi_{G,t})$ associated to the grammar G. The sets from $X_{G,1}$ to $X_{G,m}$ are associated to the m productions in P while $X_{G,t}$ contains only terminal symbols and $X_{G,0}$ initialize a sentential form. In particular, given the initial indexed state $\boldsymbol{X}_G^0$, the language is iteratively built in the last element $X_{G,t}$ of $\boldsymbol{X}_G$ such that at fixpoint $X_{G,t}^{\mathrm{fix}} = \mathscr{L}(G)$. From now on, the subscript G is dropped whenever it is clearly understood.

We introduce a barred version of the set of non-terminals N: $\bar{N} \triangleq \{\bar{A} \mid A \in N\}$ where $\bar{A}$ is the corresponding "marked" non-terminal to A. We also extend the set of variables V in order to contain marked non-terminals: $V \triangleq (N \cup \bar{N}, I^*) \cup T$. Marked non-terminals are the only symbols which can be rewritten by an indexed production.

By the previous reasoning, we know that each indexed grammar G induces a vector of π-functions $\boldsymbol{\pi}_G \triangleq (\pi_0, \pi_1, \ldots, \pi_m, \pi_t)$. For $i \in [1, m]$, we associate the

π_i-function $\pi_i \triangleq (h_i, R_i)$ with the i-th production in the enumeration of P. For all $\alpha, \beta \in V^*$ and $\sigma \in I^*$, each substitution function $h : \wp(V^*) \to \wp(V^*)$ is defined inductively as follows:

$$h_i(\varnothing) \triangleq \varnothing$$

$$h_i(\{\epsilon\}) \triangleq \{\epsilon\}$$

$$h_i(\{a\}) \triangleq \{a\} \quad \text{if } a \in T$$

$$h_i(\{(\bar{A}, \sigma)\}) \triangleq \begin{cases} \{\alpha\} & \text{if } (\bar{A}, \sigma) \in V \text{ and } [\, A \to \alpha \,]_i \in P \textit{ (Stack copy rule)} \\ \{(B, f\sigma)\} & \text{if } (\bar{A}, \sigma) \in V \text{ and } [\, A \to B_f \,]_i \in P \textit{ (Push rule)} \\ \{\beta\} & \text{if } (\bar{A}, \sigma) \in V \text{ and } [\, A_f \to \beta \,]_i \in P \textit{ (Pop rule)} \\ \{(\bar{A}, \sigma)\} & \text{otherwise} \end{cases}$$

$$h_i(\{\alpha Y\}) \triangleq h_i(\{\alpha\}) h_i(\{Y\}) \quad \text{if } \alpha \in V^+ \text{ and } Y \in V$$

$$h_i(Q) \triangleq \bigcup_{\alpha \in Q} h_i(\{\alpha\}) \quad \text{if } Q \in \wp(V^*) \text{ and } Q \neq \varnothing.$$

Intuitively, the substitution function h_i will apply the i-th production to the marked non-terminal corresponding to the non-terminal of the associated production, without checking the stack symbols, i.e., in a context-free way. The other non-terminals remain untouched.

Each regular expression R_i associated to the i-th production of P is defined as follows:

$$R_i \triangleq \begin{cases} V^*(\bar{A}, f\sigma)V^* & \text{if } [\, A_f \to \beta \,]_i \in P \textit{ (Pop rule)} \\ V^* & \text{otherwise.} \end{cases}$$

Intuitively, if the i-th indexed production $A_f \to \beta$ associated to R_i is of type $(\mathcal{Z}c)$, then only the sentential forms containing the signed non-terminal $\bar{A}$ and having f as the top symbol of its stack, will be selected from intersection.

Now, for $i \in [1, m]$, an application of $\pi_i \triangleq (h_i, R_i)$ to an indexed state corresponds to an application of the i-th indexed production.

We define the π-function π_0: its role is to mark the leftmost non-terminal of each sentential form. This marked non-terminal is the one used in the next iteration. Formally, $\pi_0 \triangleq (h_0, R_0)$ is inductively defined as follows , where here $\alpha \in V^+$:

$$h_0(\varnothing) \triangleq \varnothing$$

$$h_0(\{\epsilon\}) \triangleq \{\epsilon\}$$

$$h_0(\{Y\alpha\}) \triangleq \begin{cases} Y\, h_0(\{\alpha\}) & \text{if } Y \in T \\ (\bar{A}, \sigma)\, \texttt{unmark}(\{\alpha\}) & \text{if } Y = (A, \sigma) \text{ and } A \in N \\ (A, \sigma)\, h_0(\{\alpha\}) & \text{if } Y = (\bar{A}, \sigma) \text{ and } \bar{A} \in \bar{N} \end{cases}$$

$$h_0(Q) \triangleq \bigcup_{\alpha \in Q} h_0(\{\alpha\}), \quad \text{if } Q \in \wp(V^*) \text{ and } Q \neq \varnothing.$$

and function $\texttt{unmark} : \wp(V^*) \to \wp(V^*)$ differs from h_0 in:

$$\texttt{unmark}(\{Y\alpha\}) \triangleq \begin{cases} (A, \sigma)\, \texttt{unmark}(\{\alpha\}) & \text{if } Y = (\bar{A}, \sigma) \text{ and } \bar{A} \in \bar{N} \\ Y\, \texttt{unmark}(\{\alpha\}) & \text{otherwise.} \end{cases}$$

Intuitively, function h_0 marks the leftmost unmarked non-terminal while it unmarks every previously marked ones. The regular expression associated to h_0 is $R_0 \triangleq V^*$.

We define the π-function π_t to be applied to the last element of the state $\boldsymbol{X}$ that collects in X_t all the terminal words. Formally, $\pi_t \triangleq (h_t, R_t)$ where h_t is the identity function, namely, $h_t \triangleq id$, and $R_t \triangleq T^*$.

This leads us to the following system of equations, where $i \in [0, m]$ and $j \geq 1$:

$$\begin{cases} \boldsymbol{X}^0 & \triangleq (\{(S, \epsilon)\}, \varnothing, \ldots, \varnothing) \\ X_i^{j+1} & \triangleq \pi_i(\boldsymbol{X}^j) \\ X_t^{j+1} & \triangleq \pi_t(\boldsymbol{X}^j). \end{cases}$$

Let $(S, \epsilon) \to^n w$, with $w \in T^*$ and $n \geq 1$, be a derivation of G after n steps. We can construct a sequence of π-functions starting from the initial indexed state $\boldsymbol{X}^0$ that exactly simulate the derivations in G such that the word $w \in X_{G,t}^{2n+1}$. Indeed, an application of the i-th production in P is exactly simulated by an application of two π-functions: $\pi_{G,0}$ to mark the non-terminal used in the production and $\pi_{G,i}$ to apply the i-th index production. After $2n$ steps, an application of the π-function $\pi_{G,t}$ at step $2n+1$ yields $w \in X_{G,t}^{2n+1}$. Conversely, it is straightforward to show by induction on n that, for all $w \in T^*$, if $w \in X_{G,t}^n$, with $n \geq 1$, then there exists a derivation in G yielding w after $(n-1)/2$ steps, namely $(S, \epsilon) \to^{(n-1)/2} w$.

In order to prove that $\mathscr{L}(G)$ corresponds to the last component of the least solution of the equation $\boldsymbol{X}_G = \boldsymbol{\pi}_G(\boldsymbol{X}_G)$, it is sufficient to observe that each π-function is monotone because of $\bigcup \boldsymbol{X}$ and, furthermore, if $\boldsymbol{X}^1, \boldsymbol{X}^2, \ldots$ is a sequence in $\wp(V^*)^n$ such that $\boldsymbol{X}^1 \subseteq \boldsymbol{X}^2 \subseteq \ldots$, then $\boldsymbol{\pi}_G(\bigcup_{i=1}^{\infty} \boldsymbol{X}^i) =$

$\bigcup_{i=1}^{\infty}(\boldsymbol{\pi}_G(\boldsymbol{X}^i)$. This means that the Kleene iterates $\boldsymbol{X}^0$, $\boldsymbol{\pi}_G(\boldsymbol{X}^0)$, $\boldsymbol{\pi}_G(\boldsymbol{\pi}_G(\boldsymbol{X}^0))$, $\ldots$, starting from $\boldsymbol{X}^0 \triangleq (\{(S,\epsilon)\},\varnothing,\ldots,\varnothing)$, lead to the least fixpoint $\boldsymbol{X}^{\text{fix}}$ whose corresponding last component is $\mathscr{L}(G)$.

□

In the following, we denote the least solution of the system of equations $\boldsymbol{X} = \boldsymbol{\pi}_G(\boldsymbol{X})$ with either $\mathsf{lfp}(\boldsymbol{\pi}_G)$ or $\boldsymbol{X}^{\text{fix}}$.

Example 5.8. Consider the indexed language $L \triangleq \{a^n b^n c^n \mid n \geq 1\}$ and the indexed grammar $G \triangleq (\{S,T,A,B,C\},\{a,b,c\},\{f\},P,S)$ generating it presented in Example 5.2. Let the productions in P be enumerated as follows:

$$\begin{array}{lll} [\,S \to T\,]_1 & [\,A_f \to aA\,]_4 & [\,B_\epsilon \to b\,]_7 \\ [\,T \to T_f\,]_2 & [\,A_\epsilon \to a\,]_5 & [\,C_f \to cC\,]_8 \\ [\,T \to ABC\,]_3 & [\,B_f \to bB\,]_6 & [\,C_\epsilon \to c\,]_9 \end{array}$$

We denote a substitution as a list of replacements, e.g., $\{(\bar{S},\sigma) \to (T,\sigma)\}$ denotes the substitution h_1 defined by $h_1(\{(\bar{S},\sigma)\}) \triangleq \{(T,\sigma)\}$ and identity otherwise. Following the proof of Theorem 5.7, the fixpoint characterization of the indexed grammar of Example 5.2 is:

$$\begin{aligned}
X_0^{j+1} &= \pi_0(h_0,R_0) = h_0(V^* \cap \bigcup \boldsymbol{X}^j) \\
X_1^{j+1} &= \pi_1(h_1,R_1) = \{(\bar{S},\sigma) \to (T,\sigma)\}(V^* \cap \bigcup \boldsymbol{X}^j) \\
X_2^{j+1} &= \pi_2(h_2,R_2) = \{(\bar{T},\sigma) \to (T,f\sigma)\}(V^* \cap \bigcup \boldsymbol{X}^j) \\
X_3^{j+1} &= \pi_3(h_3,R_3) = \{(\bar{T},\sigma) \to (A,\sigma)(B,\sigma)(C,\sigma)\}(V^* \cap \bigcup \boldsymbol{X}^j) \\
X_4^{j+1} &= \pi_4(h_4,R_4) = \{(\bar{A},f\sigma) \to a(A,\sigma)\}(V^*(\bar{A},f\sigma)V^* \cap \bigcup \boldsymbol{X}^j) \\
X_5^{j+1} &= \pi_5(h_5,R_5) = \{(\bar{A},\epsilon) \to a\}(V^*(\bar{A},\epsilon)V^* \cap \bigcup \boldsymbol{X}^j) \\
X_6^{j+1} &= \pi_6(h_6,R_6) = \{(\bar{B},f\sigma) \to b(B,\sigma)\}(V^*(\bar{B},f\sigma)V^* \cap \bigcup \boldsymbol{X}^j) \\
X_7^{j+1} &= \pi_7(h_7,R_7) = \{(\bar{B},\epsilon) \to b\}(V^*(\bar{B},\epsilon)V^* \cap \bigcup \boldsymbol{X}^j) \\
X_8^{j+1} &= \pi_8(h_8,R_8) = \{(\bar{C},f\sigma) \to c(C,\sigma)\}(V^*(\bar{C},f\sigma)V^* \cap \bigcup \boldsymbol{X}^j) \\
X_9^{j+1} &= \pi_9(h_9,R_9) = \{(\bar{C},\epsilon) \to c\}(V^*(\bar{C},\epsilon)V^* \cap \bigcup \boldsymbol{X}^j) \\
X_t^{j+1} &= \pi_t(h_t,R_t) = (T^* \cap \bigcup \boldsymbol{X}^j)
\end{aligned}$$

where $\sigma \in I^*$, $\boldsymbol{X}^0 = (\{(S,\epsilon)\},\varnothing,\varnothing,\varnothing,\varnothing,\varnothing,\varnothing,\varnothing,\varnothing,\varnothing,\varnothing,\varnothing)$ and the union of all components is given by $\bigcup \boldsymbol{X}^j = \bigcup\{X_y^j \mid y \in \{0,1,2,3,4,5,6,7,8,9,t\}\}$. We have $X_t^{\text{fix}} = \mathscr{L}(G) = \{a^n b^n c^n \mid n \geq 1\}$. For example, the word "*aabbcc*" can be generated in X_t after 19 steps by the following derivations:

$$aabbcc \in \pi_t\pi_9\pi_0\pi_8\pi_0\pi_7\pi_0\pi_6\pi_0\pi_5\pi_0\pi_4\pi_0\pi_3\pi_0\pi_2\pi_0\pi_1\pi_0(\boldsymbol{X}^0).$$

Note that each π-function associated to a production $i \in [1, 9]$ is always preceded by the marking function π_0, while the final application of π_t extracts the final terminal word generated by the previous substitution functions. ■

5.3 Abstract Indexed Grammars

In this section, we want to *investigate* if the relation that exists between regular, CF, indexed and CS languages can be expressed as Galois insertions, namely, if less expressive languages can be seen as abstractions of more expressive ones.

Given a finite set of alphabet symbols Σ, we consider the complete lattice of all possible languages on Σ, namely:

$$\langle \wp(\Sigma^*), \subseteq, \cup, \cap, \Sigma^*, \varnothing \rangle.$$

Suppose that we want to model the relation between Indexed and CF languages as a GI. This means that we want to abstract an indexed language into the best (w.r.t. set inclusion) CF language that includes it. However, this is not possible since CF languages are not closed under intersection, and, therefore, the abstract domain of CF languages $\langle CF, \subseteq \rangle$ is not a Moore family. The same holds when analyzing the relation between CS and indexed languages, and the one between CF and regular languages: the families of indexed languages and of regular languages do not form a Moore family of $\langle \wp(\Sigma^*), \subseteq \rangle$, as shown in the following three examples.

Example 5.9. Consider the following family of languages:

$$\forall i \geq 0.\ L_i \triangleq \overline{\{a^i b^i\}}.$$

Each set L_i is a regular language since its complement language $\overline{L_i} = \{a^i b^i\}$ is a finite set and regular languages are closed under complement operation. This means that for every $i \geq 0$, $L_i \in REG$. By taking the intersection of all L_i we get

$$L \triangleq \bigcap_{i=0}^{\infty} L_i = \overline{\{a^n b^n \mid n \geq 0\}}.$$

The language L is CF, $L \in CF$, since it can be created from the union of several simpler languages:

$$L \triangleq \{a^i b^j \mid i > j\} \cup \{a^i b^j \mid i < j\} \cup (a \cup b)^* b (a \cup b)^* a (a \cup b)^*$$

that is, all strings of "a" followed by "b" in which the number of "a" and "b" differ, joined with all strings not of the form $a^i b^j$. The language $\{a^i b^j \mid i > j\} \in CF$ and a CF grammar generating is the following:

$$S \to aSb \mid aS \mid a.$$

Similarly $\{a^i b^j \mid i < j\} \in CF$, while $(a \cup b)^* b (a \cup b)^* a (a \cup b)^* \in REG$ since it is a regular expression. Note that we have obtained a strict CF language from an (infinite) intersection of regular languages, i.e., $L \in CF \setminus REG$. ■

Example 5.10. Consider the following two CF languages and their corresponding CF grammars:

$$\begin{array}{ll} L_1 \triangleq \{a^n b^n c^m \mid n, m \geq 0\} & L_2 \triangleq \{a^n b^m c^m \mid n, m \geq 0\} \\ S \to AC & S \to AB \\ A \to aAb \mid \epsilon & A \to aA \mid \epsilon \\ C \to cC \mid \epsilon & B \to bBc \mid \epsilon \end{array}$$

It is easy to see that their intersection corresponds to the language

$$L \triangleq L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 0\}.$$

It is well known that L is not CF, indeed it is a classical example for the Pumping Lemmata of CF languages [48], while, as we have seen in Example 5.2, L is can be described by an indexed grammar. Therefore, we can conclude that $L \in IL \setminus CF$. ■

Example 5.11. Consider the following indexed languages:

$$L_1 \triangleq \{w \in \{a, b, c\}^* \mid \#a = \#b\}$$

$$L_2 \triangleq \{w \in \{a, b, c\}^* \mid \#b = \#c\}$$

where $\#a$ means the number of symbols a in a word $w \in \Sigma^*$. L_1 can be generated by the following CF grammar:

$$S \to SS \qquad S \to aSb \qquad S \to bSa \qquad S \to c \qquad S \to \epsilon$$

and similarly for L_2, therefore, both L_1 and L_2 are also indexed languages because $CF \subset IL$. Consider the language

$$L \triangleq L_1 \cap L_2 = \{w \in \{a, b, c\}^* \mid \#a = \#b = \#c\}.$$

L is a contest-sensitive language but not indexed, namely, $L \in CS \setminus IL$. An example of a CS grammar generating L is:

$$\begin{array}{llll} S \to ABC & AC \to CA & CA \to AC & C \to c \\ S \to ABCS & BC \to CB & CB \to BC & B \to b \\ AB \to BA & BA \to AB & A \to a & \end{array}$$

Observe that we have obtained a CS language from an intersection of two indexed languages. ■

The examples above show that it is not possible to specify GIs between the domains of languages in the Chomsky hierarchy. However, this does not exclude the possibility of *approximating* the fixpoint semantics of indexed grammars by acting on the *productions* of the grammars. This corresponds to *constraining* the structures of productions or the way the memory (stack) of the productions of indexed grammars are used, by acting on the *indexed states* of the equational characterization associated. We provide abstractions of indexed grammars, namely of the mechanism used to generate the indexed languages, with the aim of transforming an indexed language into a more abstract (namely a *less expressive*) language in Chomsky's hierarchy, such as CF or REG languages.

Before showing the abstractions, let us formalize our *concrete domain* and the *concrete transition function* on it. We first define the set of *possible indexed states*.

Definition 5.12 (Set of possible indexed states). *Let $G \triangleq (N, T, I, P, S)$ be an indexed grammar, i.e., $\mathscr{L}(G) \in IL$, with m productions in P. We define*

$$\Phi_G \triangleq \wp(V^*)^n$$

as the domain of possible indexed states of G, having $n \triangleq m + 2$ components.

This means that, for every indexed state $\boldsymbol{X} \in \Phi_G$ of the indexed grammar G, we have $\boldsymbol{X} \triangleq (X_1, \ldots, X_n)$. We define a projection function on the elements of $\boldsymbol{X}$.

Definition 5.13 (Projection function). *Let $\boldsymbol{X} \triangleq (X_1, \ldots, X_n)$ be an indexed state of G. We define with $\mathtt{proj}_i^G : \Phi_G \to \boldsymbol{X}$ the projection function of the i-th element of $\boldsymbol{X} \in \Phi_G$, with $i \in [1, n]$, such that $\mathtt{proj}_i^G(\boldsymbol{X}) \triangleq X_i$.*

Note that, following the fixpoint characterization given in Theorem 5.7, for all $\boldsymbol{X} \in \Phi_G$ we have $\boldsymbol{X} \triangleq (X_0, X_1, \ldots, X_m, X_t)$, therefore $\mathtt{proj}_n^G(\boldsymbol{X}) = X_t$, i.e., the set of all terminal words $X_t \subseteq T^*$. We will omit the superscript symbol G on `proj` whenever is clearly understood.

Definition 5.14 (Concrete domain of indexed states). *We define the poset $\langle \Phi_G, \leq_\Phi \rangle$ as our concrete domain of indexed states, where the partial order $\leq_\Phi$ over Φ_G is defined as follows: $\forall \boldsymbol{X}, \boldsymbol{Y} \in \Phi_G : \ \boldsymbol{X} \leq_\Phi \boldsymbol{Y} \ \Leftrightarrow \ \forall i \in [1, n].\ X_i \subseteq Y_i$.*

It is possible to define a function on the concrete domain $\langle \Phi_G, \leq_\Phi \rangle$ that iteratively computes the language of the indexed grammar G. Thus, we define the concrete function on $\langle \Phi_G, \leq_\Phi \rangle$ called *transition relation*.

Definition 5.15 (Transition relation). *The transition relation between two indexed states $\boldsymbol{X}^i, \boldsymbol{X}^{i+1} \in \Phi_G$, with $i \geq 0$, corresponds to the application of the vector function induced by the indexed grammar G defined $\boldsymbol{\pi}_G : \Phi_G \to \Phi_G$, namely, $\boldsymbol{X}^{i+1} = \boldsymbol{\pi}_G(\boldsymbol{X}^i)$.*

In the following sections, we formalize indexed grammars abstractions as closure operators on $\langle \Phi_G, \leq_\Phi \rangle$. We start in Section 5.3.1 with a simple abstraction, stack elimination, which eliminates completely the stack of all non-terminals. Then, with the purpose of refining the abstraction, we present two other abstractions: stack limitation (Section 5.3.2), which limits the stack capacity, and stack copy limitation (Section 5.3.3), which limits stack copy productions.

5.3.1 Stack Elimination

This abstraction *removes the stack* of each non-terminal in a sentential form of an indexed grammar. Given a sentential form $\beta \in V^*$, each pair (A, σ) in β, with $A \in N$ and $\sigma \in I^*$, is replaced by (A, I^*). The idea of stack elimination is to abstract away from the stack, namely, to replace the stack with the set of all possible stacks. In other words, this corresponds to a set of sentential forms one for each possible stack value. A major consequence of applying stack elimination is that the three kinds of indexed productions (stack copy, push and pop) in an indexed grammar are turned into a single *context-free production*, i.e., of type (*3a*) by Definition 5.1. We formalize this notion through an upper closure operator ρ^E.

Definition 5.16 (Stack elimination abstraction). *We formalize the abstract domain as a closure $\rho^E : \Phi_G \to \Phi_G$ on $\langle \Phi_G, \leq_\Phi \rangle$, as follows:*

$$\rho^E(\boldsymbol{X}) \triangleq (\rho^E(X_1), \ldots, \rho^E(X_n))$$

and, with a slight abuse of notation, $\rho^E(X_i) \triangleq \{\rho^E(s_i) \mid s_i \in X_i\}$ where for $s_i \triangleq \lambda_{i1} \ldots \lambda_{iw}$ with $\lambda_{ij} \in V$ we have:

$$\rho^E(\lambda_{i1})\rho^E(\lambda_{i2} \ldots \lambda_{iw}) \triangleq \begin{cases} (A, I^*) & \text{if } \lambda_{i1} = (A, \sigma) \\ \lambda_{i1} & \text{otherwise.} \end{cases}$$

Intuitively, the stack of all non-terminal symbols is set to I^*. This means that there is no restrictions on the symbol on the top of the stack when performing a pop operation, turning push and pop productions to stack copy productions. Thus, for all $i \in [1, n]$, each sentential form in X_i containing at least one pair of non-terminal/stack symbol(s) is substituted by all the sentential forms covering all the possible combination of stack symbols, i.e., I^*.

We want to demonstrate that stack elimination is a *sound* abstraction of indexed grammars, i.e., by applying stack elimination to any indexed grammar, we get a new grammar whose generated language is still in *IL*.

Lemma 5.17. *The function $\rho^E : \Phi_G \to \Phi_G$ on the poset $\langle \Phi_G, \leq_\Phi \rangle$ is an uco. Moreover, the following holds:*

(i) $\rho^E(\mathsf{lfp}(\boldsymbol{\pi}_G)) =_\Phi \mathsf{lfp}(\boldsymbol{\pi}_G)$;

(ii) $\mathsf{lfp}(\boldsymbol{\pi}_G) \leq_\Phi \mathsf{lfp}(\boldsymbol{\pi}_G \circ \rho^E)$.

Proof. ρ^E is trivially increasing ($\boldsymbol{X} \leq_\Phi \rho^E(\boldsymbol{X})$), monotone ($\boldsymbol{X} \leq_\Phi \boldsymbol{Y} \Rightarrow \rho^E(\boldsymbol{X}) \leq_\Phi \rho^E(\boldsymbol{Y})$) and idempotent ($\rho^E(\rho^E(\boldsymbol{X})) =_\Phi \rho^E(\boldsymbol{X})$) because the resulting sentiential forms obtained by eliminating the stack of each non-terminal of the input sentential form, include the latter. Thus, it is a closure operator on $\langle \Phi_G, \leq_\Phi \rangle$. This proves also (i). To prove (ii) it is sufficient to observe that by applying ρ^E on $\boldsymbol{X}^j$ for each iteration $j \in \mathbb{N}$, i.e., setting the stack of each non-terminal to I^*, is equivalent to transforming each rule in the set of productions P of the indexed grammar G to a stack copy rule, thus ignoring the stack. Namely, rules of type $(3b)$ $A \to B_f$ turn into $A \to B$ while rules of type $(3c)$ $A_f \to \beta$ into $A \to \beta$. This means that each pair of non-terminal/stack will have an empty stack ϵ. Let $G^\sharp$ be this newly CF grammar obtained from G by stack elimination. The fixpoint characterization presented in Theorem 5.7 works also for CF grammars. Indeed, each CF grammar can be viewed as an indexed grammar having only rules of type $(3a)$. Let $\mathsf{lfp}(\boldsymbol{\pi}_{G^\sharp})$ be the fixpoint characterization of $G^\sharp$. We have $\mathsf{lfp}(\boldsymbol{\pi}_G) \leq_\Phi \mathsf{lfp}(\boldsymbol{\pi}_{G^\sharp}) = \mathsf{lfp}(\boldsymbol{\pi}_G \circ \rho^E)$. □

Lemmas 5.17 allows us to prove the soundness of the stack elimination abstraction by showing that all languages obtained by stack elimination from an indexed grammar are an over approximation of their original indexed languages.

Theorem 5.18. *Let $L \in IL$ and G be an indexed grammar such that $\mathscr{L}(G) = L$. Let $L^\sharp \triangleq \mathtt{proj}_n(\mathsf{lfp}(\boldsymbol{\pi}_G \circ \rho^E))$ and $G^\sharp$ be an indexed grammar such that $\mathscr{L}(G^\sharp) = L^\sharp$. Then $L \subseteq L^\sharp$.*

Proof.

$$\begin{aligned} L = \mathscr{L}(G) &= \mathtt{proj}_n(\mathsf{lfp}(\boldsymbol{\pi}_G)) && \text{[by Theorem 5.7]} \\ &\subseteq \mathtt{proj}_n(\mathsf{lfp}(\boldsymbol{\pi}_G \circ \rho^E)) && \text{[by Lemma 5.17]} \\ &= \mathscr{L}(G^\sharp) = L^\sharp. \end{aligned}$$

□

The loss of precision, i.e., the less expressive language w.r.t. subset inclusion, obtained thorough ρ^E, is due to the fact that, when eliminating the stack, an indexed grammar can no longer count or store occurrences of an index symbol, thus it is reduced to a CF grammar. Moreover, it turns out that if the original indexed grammar G is such that $\mathscr{L}(G) \in IL$ but $\mathscr{L}(G) \notin CF$ then $\mathscr{L}(G^\sharp)$ is a CF language but not indexed.

Proposition 5.19. *Let G be any indexed grammar such that $\mathscr{L}(G) \in IL \setminus CF$. Let $G^\sharp$ be the indexed grammar obtained from G by the stack elimination abstraction ρ^E. Then $\mathscr{L}(G^\sharp) \notin IL \setminus CF$.*

Proof. The proof is straightforward by observing that $\mathscr{L}(G) \in IL \setminus CF$ implies that G uses at least one stack memory of a non-terminal in order to generate the terminal symbols, i.e., G has at least one production rule of type (*3b*) and one of type (*3c*), otherwise it is a CF grammar. Then, clearly by applying ρ^E, all the productions in G turns into productions rules of type (*3a*). Therefore, the obtained grammar $G^\sharp$ is CF and its generated language $\mathscr{L}(G^\sharp)$ is CF. □

Example 5.20. We have seen in Example 5.2 that the language $L \triangleq \{a^n b^n c^n \mid n \geq 1\}$ is an indexed language but not CF, namely, $L \in IL \setminus CF$. If we apply stack elimination abstraction on the induced vector function $\boldsymbol{\pi}_G$ at fixpoint we obtain a new language $\mathtt{proj}_n(\mathsf{lfp}(\boldsymbol{\pi}_G \circ \rho^E))$ which can be generated by a new grammar $G^\sharp$ with the following productions:

$$\begin{array}{lll} S \to T & A \to aA & B \to b \\ T \to T & A \to a & C \to cC \\ T \to ABC & B \to bB & C \to c \end{array}$$

Note that this grammar corresponds exactly to the indexed grammar in Example 5.2 by "manually" removing the stack from the productions. It is easy to note that the language generated from $G^\sharp$ is

$$\mathscr{L}(G^\sharp) = \mathtt{proj}_n(\mathsf{lfp}(\boldsymbol{\pi}_G \circ \rho^E)) = \{a^* b^* c^*\}$$

which is a regular expression, thus, a regular language, $\mathscr{L}(G^\sharp) \in REG$ which means also that $\mathscr{L}(G^\sharp) \in CF$. Moreover, since $\{a^n b^n c^n \mid n \geq 1\} \subset \{a^* b^* c^*\}$, we have $\mathscr{L}(G) \subset \mathscr{L}(G^\sharp)$. ■

Although some examples may be deceiving, in general it is not true that all indexed languages become regular by stack elimination. Indeed, indexed grammars could contain *context-free characteristic* rules that do not affect stacks hence, after stack elimination, still remain strict CF, turning the language to a CF language and not regular. The following is an example of such an indexed grammar.

Example 5.21. The language $L \triangleq \{a^n b^n c^n \mid n \geq 1\}$ can be also generated by a new indexed grammar $\hat{G}$ having the following productions in $\hat{P}$:

$$\begin{array}{ll} S \to aS_f c & T_f \to Tb \\ S \to T & T_\epsilon \to \epsilon \end{array}$$

If we apply stack elimination on $\boldsymbol{\pi}_{\hat{G}}$, at fixpoint we obtain the language

$$L^\sharp \triangleq \mathtt{proj}_n(\mathsf{lfp}(\boldsymbol{\pi}_{\hat{G}} \circ \rho^E)) = \{a^n b^* c^n \mid n \geq 1\}$$

and, clearly, $L \subset L^\sharp$, $L^\sharp \in CF$ but $L^\sharp \notin REG$. Note that, in this case, the stack elimination abstraction acts on the stack of b's symbol only, since a's and c's symbols are handled by CF-like production in $S \to aS_f c$. This explains the transition from b^n to b^* and not the same for a^n and c^n. ■

It is indeed obvious to observe that stack elimination produces *precisely* the class of CF languages.

Proposition 5.22. *Given any language $L^\sharp \in CF$ and a CF grammar $G^\sharp$ generating it, there exists an indexed grammar G such that:*

$$\mathsf{lfp}(\boldsymbol{\pi}_{G^\sharp}) =_\Phi \mathsf{lfp}(\boldsymbol{\pi}_G \circ \rho^E) =_\Phi \mathsf{lfp}(\boldsymbol{\pi}_G).$$

Proof. Let $G^\sharp \triangleq (N, T, P, S \in N)$. Then, define $G \triangleq (N, T, P, \varnothing, S)$. Because G does not use stacks, ρ^E corresponds to the identity function on it. Then, clearly we have $\mathsf{lfp}(\boldsymbol{\pi}_{G^\sharp}) =_\Phi \mathsf{lfp}(\boldsymbol{\pi}_G \circ \rho^E) =_\Phi \mathsf{lfp}(\boldsymbol{\pi}_G)$. □

One possible application of stack elimination abstraction concerns the problem of checking whether two given indexed languages $L_1 \in IL$ and $L_2 \in IL$ are *disjoint*, i.e., $L_1 \cap L_2 \stackrel{?}{=} \varnothing$. This is a fundamental language theoretical problem derived from the more common problem of *CF disjointess*. It is of interest in many practical tasks that call for some kind of automated reasoning about programs. This can be because program behavior is modeled using indexed or CF languages, as in software verification approaches that try to capture a program's control flow as a (nested pushdown-system) path language. The problem of indexed and CF disjointness is in general undecidable [49]. However, if we can compute a *regular abstraction* $L_1^\sharp \in REG$ of L_1 (resp. $L_2^\sharp \in REG$ of L_2) such that $L_1 \subseteq L_1^\sharp$ (resp. $L_2 \subseteq L_2^\sharp$) and $L_1^\sharp \cap L_2 = \varnothing$ (resp. $L_1 \cap L_2^\sharp = \varnothing$), then the answer is true, they are disjoint. Indeed, the language representing the intersection between an indexed (resp. CF) language and a regular language, is indexed (resp. CF), and the emptiness problem in IL (resp. CF) is decidable.

Example 5.23. Consider the following two languages $L_1 \triangleq \{a^n b^n c^n \mid n \geq 1\}$ and $L_2 \triangleq \{(ac)^n (ba)^n \mid n \geq 1\}$. We want to check whether $L_1 \cap L_2 \stackrel{?}{=} \varnothing$. Clearly $L_1 \in IL \setminus CF$ and $L_2 \in CF \setminus REG$. This means that the problem is in general undecidable. However, given the indexed grammar G_1 of Example 5.2 generating L_1, we can apply the stack elimination abstraction ρ^E on its fixpoint characterization in order to get an over approximation of it. Indeed, we know that

$$L_1^\sharp \triangleq \mathtt{proj}_n(\mathsf{lfp}(\boldsymbol{\pi}_{G_1} \circ \rho^E)) = \{a^* b^* c^*\}$$

and $L_1 \subset L_1^\sharp \in REG$. The check $L_1^\sharp \cap L_2 \stackrel{?}{=} \varnothing$ is decidable since $L_1^\sharp$ is a regular language and the answer is true. Hence, we can conclude that L_1 and L_2 are disjoint, i.e., $L_1 \cap L_2 = \varnothing$. ■

5.3.2 Stack Limitation

The idea of stack limitation abstraction is to *limit the number of symbols on the stack* of each non-terminal by a constant $k \geq 0$. This means that each stack can contain at most k symbols and all others $k+1$ symbols pushed on to the stack will be discarded. We want to formalize this new abstraction and checking if it is a sound abstraction of indexed grammars.

Definition 5.24 (Stack limitation abstraction). *We formalize the abstract domain as an upper closure operator ρ_k^L, with $k \geq 0$, on the concrete domain $\langle \Phi_G, \leq_\Phi \rangle$. We define $\rho_k^L : \Phi_G \to \Phi_G$ as follows:*

$$\rho_k^L(\boldsymbol{X}) \triangleq (\rho_k^L(X_1), \ldots, \rho_k^L(X_n))$$

and, with a slight abuse of notation, $\rho_k^L(X_i) \triangleq \{\rho_k^L(s_i) \mid s_i \in X_i\}$ where for $s_i \triangleq \lambda_{i1} \ldots \lambda_{iw}$ with $\lambda_{ij} \in V$ we have:

$$\rho_k^L(\lambda_{i1})\rho_k^L(\lambda_{i2} \ldots \lambda_{iw}) \triangleq \begin{cases} (A, \hat{\sigma}) & \text{if } \lambda_{i1} \triangleq (A, \sigma) \text{ and } |\sigma| > k \\ \lambda_{i1} & \text{otherwise} \end{cases}$$

where $|\sigma| = z \in \mathbb{N}$ if $\sigma = q_z \ldots q_{k+1} q_k \ldots q_1$ (the empty stack has zero cardinality, i.e., $|\epsilon| = 0$) with $\sigma \in I^*$, and for $1 \leq i \leq z$, $q_i \in I$, q_z top symbol and $\hat{\sigma} = q_k \ldots q_1$.

Intuitively, by function ρ_k^L, the stack of each non-terminal is limited to k symbols and each additional push of others symbols will be discarded. Observe that this technique corresponds to limiting push productions only.

Lemma 5.25. *The function ρ_k^L on the poset $\langle \Phi_G, \leq_\Phi \rangle$ is an uco. Moreover, we have $\rho_k^L(\mathsf{lfp}(\boldsymbol{\pi}_G)) =_\Phi \mathsf{lfp}(\boldsymbol{\pi}_G)$.*

Proof. The proof is similar to the proof of Lemma 5.17. □

In the following theorem we prove that stack limitation, as defined by the uco ρ_k^L, is *not* a sound abstraction, namely, at fixpoint, the language generated is not always an over approximation of the original concrete language.

Theorem 5.26. $\mathsf{lfp}(\boldsymbol{\pi}_G) \not\leq_\Phi \mathsf{lfp}(\boldsymbol{\pi}_G \circ \rho_k^L)$.

Proof. The proof is made by providing a counterexample. Consider the language $L \triangleq \{a^n b^n c^n \mid n \geq 1\}$ in Example 5.2. It has only one push production: $T \to T_f$. Therefore, after stack limitation, each stack can contain at most k index symbols of f. This corresponds to the following family of languages:

$$\forall k \geq 0.\ L_k \triangleq \{a^n b^n c^n \mid 1 \leq n \leq k+1\}.$$

For all $k \geq 0$, the language L_k is a regular language since each family contains a finite number of words: for $k = 0$ then $\{abc\}$, $k = 1$ then $\{abc, aabbcc\}$, ... Moreover, each L_k is not an over approximation of the original language since $L \not\subseteq L_k$, this leads to $\mathsf{lfp}(\boldsymbol{\pi}_G) \not\leq_\Phi \mathsf{lfp}(\boldsymbol{\pi}_G \circ \rho_k^L)$. □

Observe that the infinite intersection of all family of languages L_k obtained by stack limitation from a language L, corresponds to L_0, namely $\bigcap_{k=0}^{\infty} L_k = L_0$, while the infinite union of all L_k is a superset of the original language L, namely $L \subseteq \bigcup_{k=0}^{\infty} L_k$. At first glance, the family of languages generated from a language non-stack-limited is regular, as the previous example showed but, in general, this is not always true: the next example shows a counterexample, similar to Example 5.21:

Example 5.27. The language $L \triangleq \{a^n b^n c^n \mid n \geq 1\}$ could be generated also by the following indexed grammar:

$$\begin{array}{ll} S \to aS_f c & T_f \to Tb \\ S \to T & T_\epsilon \to \epsilon \end{array}$$

If we apply stack limitation we get the following family of languages:

$$\forall k \geq 1.\ \ L_k \triangleq \{a^n b^m c^n \mid n \geq 1 \ \wedge \ m \leq k\}$$

Note that for all $k \geq 1$, $L_k \in CF$ while for $k = 0$, $L_0 \triangleq \{\epsilon\}$ and the empty word ϵ is not accepted by L. ■

We can force the soundness of the stack limitation abstraction by modifying the uco ρ_k^L as follows, obtaining the new uco ρ_k^{LE}.

Definition 5.28 (Sound stack limitation abstraction). *We define* $\rho_k^{LE} : \Phi_G \to \Phi_G$ *as follows:*

$$\rho_k^{LE}(\boldsymbol{X}) \triangleq \cdots = \rho_k^{LE}(\lambda_{ij}) = \begin{cases} (A, I^*) & \text{if } \lambda_{i1} = (A, \sigma) \text{ and } k = 0 \\ (A, \hat{\sigma} I^*) & \text{if } \lambda_{i1} = (A, \sigma) \text{ and } 0 < k < |\sigma| \\ \lambda_{i1} & \text{otherwise} \end{cases}$$

where if $|\sigma| = z \in \mathbb{N}$ *then* $\sigma = q_1 \dots q_k q_{k+1} \dots q_z$ *where* q_1 *top symbol and* $\hat{\sigma} = q_1 \dots q_k$.

Intuitively, ρ_k^{LE} keeps the first k symbols on the top of the stack of each non-terminal and it removes the remaining symbols after the k-index. This means that, after popping out of the stack the first k symbols, we obtain the same result of the stack elimination abstraction. This abstraction is a refinement of the stack elimination abstraction since, in the worst case ($k = 0$), it produces exactly the same language as ρ^E, namely $\rho_0^{LE} = \rho^E$. We now prove the soundness of this new abstraction.

Lemma 5.29. $\mathsf{lfp}(\boldsymbol{\pi}_G) \leq_\Phi \mathsf{lfp}(\boldsymbol{\pi}_G \circ \rho_k^{LE})$.

Proof. The proof is a straightforward consequence of the proof of Lemma 5.17. □

Theorem 5.30. *Let $L \in IL$ and G be an indexed grammar such that $\mathscr{L}(G) = L$. Let $L^\sharp \triangleq \mathtt{proj}_n(\mathsf{lfp}(\boldsymbol{\pi}_G \circ \rho_k^{LE}))$ and $G^\sharp$ be an indexed grammar such that $\mathscr{L}(G^\sharp) = L^\sharp$. Then $L \subseteq L^\sharp$.*

Proof. Immediate by Lemma 5.29. □

Example 5.31. Consider the indexed language $L \triangleq \{a^n b^n c^n \mid n \geq 1\}$ and the indexed grammar generating it presented in Example 5.2. If we apply the sound stack limitation abstraction for some $k > 0$ on the induced vector function $\boldsymbol{\pi}_G$, we obtain the following language:

$$\mathtt{proj}_n(\mathsf{lfp}(\boldsymbol{\pi}_G \circ \rho_k^{LE})) = \{a^k a^* b^k b^* c^k c^*\}$$

which is clearly a regular language since k is fixed. However, this new language is slightly better, in terms of language inclusion, than the language obtained in Example 5.20, indeed:

$$\mathtt{proj}_n(\mathsf{lfp}(\boldsymbol{\pi}_G \circ \rho_k^{LE})) = \{a^k a^* b^k b^* c^k c^*\} \subset \{a^* b^* c^*\} = \mathtt{proj}_n(\mathsf{lfp}(\boldsymbol{\pi}_G \circ \rho^E)).$$

■

5.3.3 Stack Copy Limitation

Stack copy limitation *limits the copy of the stack*, from the right-side of a production, to a finite number of non-terminals symbols in a given set $H \subseteq N$ where N is the set of all non-terminal symbols of G. The contents of the other stacks are set to I^* meaning that you can do push and pop operations with no limits, similarly to ρ^E. For example, if $H = \{A\}$ then a production $T \to ABC$ corresponds to the derivation $(T, \sigma) \to (A, \sigma)(B, I^*)(C, I^*)$. Note that only the stack of $A \in H$ has been copied, while for $B, C \notin H$ the stack has been removed. As done in the previous sections, we define a new uco ρ_H^C for the stack copy abstraction.

Definition 5.32 (Stack copy limitation abstraction). *Given an indexed grammar $G = (N, T, I, P, S)$, we formalize the abstract domain as an upper closure operator ρ^C_H on the concrete domain $\langle \Phi_G, \leq_\Phi \rangle$ with $H \subseteq N$. We define $\rho^C_H : \Phi_G \to \Phi_G$ as:*

$$\rho^C_H(\boldsymbol{X}) \triangleq (\rho^C_H(X_1), \ldots, \rho^C_H(X_n))$$

and, with a slight abuse of notation, $\rho^C_H(X_i) \triangleq \{\rho^C_H(s_i) \mid s_i \in X_i\}$ where for $s_i \triangleq \lambda_{i1} \ldots \lambda_{iw}$ with $\lambda_{ij} \in V$ we have:

$$\rho^C_H(\lambda_{i1})\rho^C_H(\lambda_{i2} \ldots \lambda_{iw}) \triangleq \begin{cases} (A, I^*) & \text{if } \lambda_{i1} = (A, \sigma) \text{ and } A \notin H \\ \lambda_{i1} & \text{otherwise.} \end{cases}$$

Intuitively, the function ρ^C_H eliminates the stack of only a restricted set of non-terminals, namely those not in the set H, while for all non-terminals in H the stack will be copied and so all the indices symbols on it still remain. The following proposition is a straightforward consequence of Definition 5.32 of ρ^C_H.

Proposition 5.33. *For every indexed grammar $G = (N, T, I, P, S)$ and $H \subseteq N$, if $H = N$ then $\rho^C_H = id$, while if $H = \varnothing$ then $\rho^C_H = \rho^E$.*

By allowing no stack copy limitation, which means $H = N$, then the abstraction has no effect, while by rejecting all the stack copy, which means $H = \varnothing$, then we obtain stack elimination.

We show that ρ^C_H is a sound abstraction of indexed grammars.

Lemma 5.34. *Given $H \subseteq N$, the function $\rho^C_H : \Phi_G \to \Phi_G$ on the poset $\langle \Phi_G, \leq_\Phi \rangle$ is an uco. Moreover, the following holds:*

(i) $\rho^C_H(\mathsf{lfp}(\boldsymbol{\pi}_G)) \;=_\Phi\; \mathsf{lfp}(\boldsymbol{\pi}_G)$;

(ii) $\mathsf{lfp}(\boldsymbol{\pi}_G) \;\leq_\Phi\; \mathsf{lfp}(\boldsymbol{\pi}_G \circ \rho^C_H)$.

Proof. Follows trivially by the definition of ρ^C_H. □

Theorem 5.35. *Let $L \in IL$ and G be an indexed grammar such that $\mathscr{L}(G) = L$. Let $L^\sharp = \mathtt{proj}_n(\mathsf{lfp}(\boldsymbol{\pi}_G \circ \rho^C_H))$ and $G^\sharp$ be an indexed grammar such that $\mathscr{L}(G^\sharp) = L^\sharp$. Then $L \subseteq L^\sharp$.*

As expected, the quality of this abstraction, in terms of subset inclusion between languages, may be better than stack elimination and stack limitation, depending on which non-terminals form the set $H \subseteq N$, as shown in the following example.

Example 5.36. Consider the language $L \triangleq \{a^n b^n c^n \mid n \geq 1\}$ and let $H \triangleq \{A\}$. Then, we obtain:

$$L^\sharp_1 \triangleq \mathtt{proj}_n(\mathsf{lfp}(\boldsymbol{\pi}_G \circ \rho^C_H)) = \{a^n b^* c^* \mid n \geq 1\}.$$

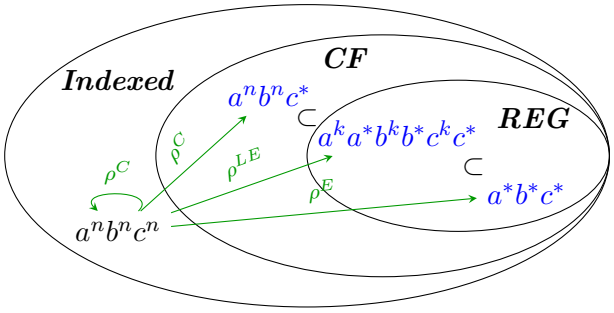


Figure 5.2: Three sound abstractions of indexed grammars presented in Section 5.3 and applied to the indexed language $\{a^n b^n c^n \mid n \geq 1\}$

Observe that $L_1^\sharp \in REG$ and $L \subset L_1^\sharp$, indeed, if H contains one of the three non-terminals then stack copy limitation is equivalent eliminate the stack of all non-terminals in H. However, if we set $H \triangleq \{A, B\}$, then by stack copy limitation we obtain

$$L_2^\sharp \triangleq \mathtt{proj}_n(\mathsf{lfp}(\boldsymbol{\pi}_G \circ \rho_H^C)) = \{a^n b^n c^* \mid n \geq 1\}.$$

Note that $L \subset L_2^\sharp \subset L_1^\sharp$, $L_2^\sharp \in CF$ and $L_2^\sharp \notin REG$. ■

We conclude by showing in Figure 5.2 the three sound abstractions presented in this section applied to the indexed language $\{a^n b^n c^n \mid n \geq 1\}$.

5.4 (In)Completeness of Abstractions of Indexed Grammars

The importance of completeness in static program analysis is evident: a program complete for a store abstraction means that no imprecision will arise in the output of the static analysis using that specific store abstraction. Conversely, incompleteness represents an, although correct, imperfect analysis in terms of precision. In this section, we study the counterpart consequences of incompleteness in language abstractions. We provide a first step into this direction, by constructing the separation class of languages between indexed and CF languages using incompleteness of the stack elimination abstraction ρ^E. In particular, we have seen in Section 5.3.1 that the class of all CF languages CF corresponds to the stack elimination abstraction ρ^E over indexed languages IL, namely, stack elimination abstraction over the fixpoint characterization of indexed grammars generating the indexed languages in IL. We want to define the separation result between IL and CF as the set of all indexed languages such that, for every

indexed grammar describing them, its fixpoint stack elimination abstraction is incomplete.

Let IG be the set of all possible indexed grammars, such that, for all $G = (N, T, I, P, S) \in IG$, $\mathscr{L}(G) \in IL$. We start with the definition of completeness of an abstraction of indexed grammars.

Definition 5.37 (Completeness of an indexed grammar). *An indexed grammar $G \in IG$ is complete for an abstraction of indexed states $\rho \in \mathrm{uco}(\Phi_G)$ when*

$$\rho(\mathsf{lfp}(\boldsymbol{\pi}_G)) =_{\Phi} \mathsf{lfp}(\boldsymbol{\pi}_G \circ \rho).$$

The collection of all indexed grammars complete for a given abstraction ρ forms the completeness class $\mathbb{C} : \mathrm{uco}(\Phi_G) \to \wp(IG)$.

Definition 5.38 (The class of complete indexed grammars). *Given an abstraction of indexed grammars $\rho \in \mathrm{uco}(\Phi_G)$, its completeness class $\mathbb{C}(\rho) \subseteq IG$ is defined as:*

$$\mathbb{C}(\rho) \triangleq \{G \in IG \mid \rho(\mathsf{lfp}(\boldsymbol{\pi}_G)) =_{\Phi} \mathsf{lfp}(\boldsymbol{\pi}_G \circ \rho)\}.$$

Let us recall that for all $L \in CF$ there exists an (countable) infinite number of possible indexed grammar $G \in IG$ generating it. Moreover, G could either use stacks, i.e., all the three rules in Definition 5.1, to generate terminal symbols, or only production rules of type (*3a*). This means that abstractions of indexed grammars could have different outputs depending on G and, therefore, completeness is strictly related to the structure of G, as shown in the following example.

Example 5.39. Consider the language $L \triangleq \{a^n b^n \mid n \geq 1\}$. Clearly $L \in CF$ since it can be generated by the following CF grammar $G_1 \triangleq (\{S\}, \{a, b\}, P_1, S)$ having the following two productions in P_1:

$$S \to aSb$$
$$S \to \epsilon.$$

Note that G_1 can be considered as an indexed grammar with no indexed symbols, i.e., $G_1 \triangleq (\{S\}, \{a, b\}, \varnothing, P_1, S)$ thus maintaining the stack of S empty. Let $G_2 \triangleq (\{S, A, B\}, \{a, b\}, \{f\}, P_2, S)$ be an indexed grammar where P_2 contains the following productions:

$$\begin{array}{ll} S \to S_f & S \to AB \\ A_f \to aA & A_\epsilon \to a \\ B_f \to bB & B_\epsilon \to b \end{array}$$

G_2 is another grammar generating L, indeed $\mathscr{L}(G_2) = \mathscr{L}(G_1) = \{a^n b^n \mid n \geq 1\}$. However, the stack elimination abstraction ρ^E on the fixpoint characterizations of G_1 and G_2 gives different results:

$$\begin{aligned}\mathtt{proj}_n(\mathsf{lfp}(\boldsymbol{\pi}_{G_1} \circ \rho^E)) &= \{a^n b^n \mid n \geq 1\}\\ \mathtt{proj}_n(\mathsf{lfp}(\boldsymbol{\pi}_{G_2} \circ \rho^E)) &= \{a^* b^*\}.\end{aligned}$$

Note that ρ^E is complete for G_1 because

$$\begin{aligned}\{a^n b^n \mid n \geq 1\} &= \mathtt{proj}_n(\mathsf{lfp}(\boldsymbol{\pi}_{G_1} \circ \rho^E))\\ &= \mathtt{proj}_n(\rho^E(\mathsf{lfp}(\boldsymbol{\pi}_{G_1})))\\ &= \mathtt{proj}_n(\mathsf{lfp}(\boldsymbol{\pi}_{G_1})) \qquad \text{[by Lemma 5.17]}\\ &= \{a^n b^n \mid n \geq 1\}\end{aligned}$$

while it is incomplete for G_2:

$$\{a^n b^n \mid n \geq 1\} = \mathtt{proj}_n(\mathsf{lfp}(\boldsymbol{\pi}_{G_2})) \subset \mathtt{proj}_n(\mathsf{lfp}(\boldsymbol{\pi}_{G_2} \circ \rho^E)) = \{a^* b^*\}.$$

Therefore, we can conclude that $G_1 \in \mathbb{C}(\rho^E)$ and $G_2 \notin \mathbb{C}(\rho^E)$. Intuitively, this incompleteness originates from the use of a stack memory in G_2 in order to generate terminal symbols, although the language is in CF. Conversely, G_1 uses only rules of type (3a), therefore, in this case, stack elimination abstraction is useless because it corresponds to the identity function. ■

The strong influence of the structure of an indexed grammar G on the completeness of the abstraction ρ, remarks the intensional aspects of G, i.e., the way the productions in G compute the language, namely, its semantics. This means that the calculational way of any indexed grammar is concerned with intensional properties not just because semantically equivalent grammars (i.e., generating the same language) may exhibit different properties, but also because semantically different grammars may appear identical when abstracted. This phenomenon is very common in program analysis where the way a program is written plays a central role for the completeness of an abstraction of stores, as are the productions in an indexed grammar.

Example 5.40. Let $\Sigma \triangleq \{a, b\}$. Consider the following two languages

$$\begin{aligned}L_1 &\triangleq \{ww \mid w \in \Sigma^*\}\\ L_2 &\triangleq \{www \mid w \in \Sigma^*\}.\end{aligned}$$

Intuitively, L_1 contains all words $w \in \Sigma^*$ repeated two times, while repeated three times in L_2. Both L_1 and L_2 are indexed languages, $L_1, L_2 \in IL$. Indeed,

a possible indexed grammar generating L_1 is $G_1 \triangleq (\{S,T\},\{a,b\},\{f,g\},P_1,S)$ having the following productions in P_1:

$$\begin{array}{ll} S \to S_g & T_f \to aT \\ S \to S_f & T_g \to bT \\ S \to TT & T_\epsilon \to \epsilon \end{array}$$

while $G_2 \triangleq (\{S,T\},\{a,b\},\{f,g\},P_2,S)$ is a possible indexed grammar for L_2 with the following productions in P_2:

$$\begin{array}{ll} S \to S_g & T_f \to aT \\ S \to S_f & T_g \to bT \\ S \to TTT & T_\epsilon \to \epsilon \end{array}$$

The stack elimination abstraction on the fixpoint characterization of G_1 and G_2 gives the following result:

$$\mathtt{proj}_n(\mathsf{lfp}(\boldsymbol{\pi}_{G_1} \circ \rho^E)) = \mathtt{proj}_n(\mathsf{lfp}(\boldsymbol{\pi}_{G_2} \circ \rho^E)) = \Sigma^*.$$

which proves that $G_1, G_2 \notin \mathbb{C}(\rho^E)$. Note that G_1 and G_2 exhibit different semantics, indeed they generate two different languages

$$\mathscr{L}(G_1) = \{ww \mid w \in \Sigma^*\} \neq \{www \mid w \in \Sigma^*\} = \mathscr{L}(G_2)$$

although the stack elimination abstraction produces the same (regular) language. ■

We can now define the *separation class* between indexed languages and CF languages as *instances of the incompleteness of stack elimination abstraction*, i.e., the complement set of $\mathbb{C}(\rho^E)$:

$$\overline{\mathbb{C}(\rho^E)} = \{G \in IG \mid \rho^E(\mathsf{lfp}(\boldsymbol{\pi}_G)) \neq_\Phi \mathsf{lfp}(\boldsymbol{\pi}_G \circ \rho^E)\}.$$

Definition 5.41 (Separation class between IL and CF). *We define* $\mathsf{Sep}_{\rho^E}^{IL-CF}$ *as the separation class between the class of all indexed languages IL and the class of all CF languages CF, as:*

$$\mathsf{Sep}_{\rho^E}^{IL-CF} \triangleq \{L \in IL \mid \forall G \in IG.\ \mathscr{L}(G) = L \ \Rightarrow\ G \in \overline{\mathbb{C}(\rho^E)}\}.$$

Intuitively, $\mathsf{Sep}_{\rho^E}^{IL-CF}$ is the set of all indexed languages $L \in IL$ such that, the fixpoint abstraction by stack elimination ρ^E of each indexed grammar $G \in IG$ such that $\mathscr{L}(G) = L$, is incomplete for ρ^E.

Theorem 5.42. $L \in IL \smallsetminus CF \ \Leftrightarrow\ L \in \mathsf{Sep}_{\rho^E}^{IL-CF}$

Proof. ($\Rightarrow$) Suppose that $L \in IL \setminus CF$ and let $G \in IG$ be an indexed grammar such that $\mathscr{L}(G) = L$. Then we have:

$$\begin{aligned}
\mathscr{L}(G) &= \mathtt{proj}_n(\mathsf{lfp}(\boldsymbol{\pi}_G)) && \text{[by Theorem 5.7]}\\
&= \mathtt{proj}_n(\rho^E(\mathsf{lfp}(\boldsymbol{\pi}_G))) && \text{[by Lemma 5.17]}\\
&\subset \mathtt{proj}_n(\mathsf{lfp}(\boldsymbol{\pi}_G \circ \rho^E)) && \text{[by Theorem 5.18 and Proposition 5.19]}
\end{aligned}$$

Since by definition of ρ^E:

$$\mathtt{proj}_n(\rho^E(\mathsf{lfp}(\boldsymbol{\pi}_G))) \subset \mathtt{proj}_n(\mathsf{lfp}(\boldsymbol{\pi}_G \circ \rho^E)) \;\Rightarrow\; \rho^E(\mathsf{lfp}(\boldsymbol{\pi}_G)) \neq_{\Phi} \mathsf{lfp}(\boldsymbol{\pi}_G \circ \rho^E)$$

this implies that $G \in \overline{\mathbb{C}(\rho^E)}$. Therefore, we can conclude $L \in \mathsf{Sep}_{\rho^E}^{IL-CF}$.

($\Leftarrow$) By contradiction, suppose that $L \in \mathsf{Sep}_{\rho^E}^{IL-CF}$ and $L \notin IL \setminus CF$. Because $L \in \mathsf{Sep}_{\rho^E}^{IL-CF}$ then L cannot be outside IL otherwise $L \notin \mathsf{Sep}_{\rho^E}^{IL-CF}$. Therefore, L must be a CF language, i.e., $L \in CF$. This implies that there exists a CF grammar G_{CF} such that $\mathscr{L}(G_{\text{CF}}) = L$. By Proposition 5.22, there exists an indexed grammar $G \in IL$ such that $\mathsf{lfp}(\boldsymbol{\pi}_{G_{\text{CF}}}) =_{\Phi} \mathsf{lfp}(\boldsymbol{\pi}_G \circ \rho^E) =_{\Phi} \mathsf{lfp}(\boldsymbol{\pi}_G)$. But this implies that $\mathscr{L}(G) = L$ and $G \in \mathbb{C}(\rho^E)$, hence $L \notin \mathsf{Sep}_{\rho^E}^{IL-CF}$ which contradicts the assumption. We can conclude that L must be in $IL \setminus CF$. □

Theorem 5.42 provides a further insight into the structure of Chomsky's hierarchy of formal languages. It establishes the *equivalence* between the set of all indexed languages not representable with a CF grammar, and the set of all indexed languages for which no indexed grammar generating them is complete for the stack elimination abstraction. Therefore, this equivalence formalizes the separation result $IL \setminus CF$ as an instance of the incompleteness of ρ^E. This means that, in order to prove that $L \notin CF$ but $L \in IL$, it is sufficient to show that $L \in \mathsf{Sep}_{\rho^E}^{IL-CF}$. Toward showing that Chomsky's hierarchy can be understood by abstract interpretation between grammars, this result demonstrates one such step between indexed grammars and context-free ones.

Chapter 6

RELATED WORK

Completeness is a well known notion in static program analysis by abstract interpretation and the classes of complete programs for a given abstraction have been recently studied [8,9,44]. In [44] the authors introduce the notion of completeness class as the set of all programs that are complete with regard to a given abstract domain, together with a sound stratified deductive system for proving the completeness of program analysis over an abstract domain. Bruni et al. [8] introduced the concept of completeness cliques as the set of equivalent programs that are complete with regard to an abstract domain. In particular, they prove that there exists a total recursive function that transforms any complete program into a semantically equivalent but incomplete one for a given abstraction. This formalization resembles the definition of complexity clique defined by Asperti in [4]. The first attempt to weaken the notion of completeness in abstract interpretation has been recently introduced by Bruni et al. in [9]. Here the authors introduced the notion of local completeness, that is, completeness among certain program traces. They provided a logical proof system that combines over and under-approximations of programs behaviors. However, these works do not distinguish among incompleteness results: an analysis is either complete or incomplete but no further formalization is available for reasoning about the level of imprecision associated to an incomplete analysis. Our approach can be considered as a further weakening of local completeness, as our aim is to be able to measure and reason about the imprecision induced by program analysis.

Some papers have addressed the problem of measuring the imprecision of abstract interpretation and static analysis. Among the earliest, Crazzolara [31] proposes the use of quasi-metric spaces, instead of partial orders, as an alternative to define a new framework for abstract interpretation of programs, by applying the Banach's contraction principle as an alternative for Knaster-Tarsky's fixpoint theorem. Conversely, our approach considers quasi-metrics as

external measures and we follow the classical framework of abstract interpretation in [26] and [27] for the concrete collecting and abstract semantics. Other notions of distance have been employed as a measure of imprecision in program analysis. In [13] the authors propose distances in logic programming domains for measuring the precision of analysis, while in [56] the authors introduce the notion of pseudo-distance, a weaker form of metric similar to quasi-metric definition, as an external measure function to quantifying the relative loss of precision induced by numerical abstract domains. Sotin in [71] defines measures in $\mathbb{R}^n$ that allow us to quantify the difference in precision between two abstract values of a numeric domain, by comparing the size of their concretizations. This is applied to guessing the most appropriate domain to analyze a program, by under-approximating the potentially visited states via random testing and comparing the precision with which different domains would approximate those states. Di Pierro et al. [37] propose a notion of probabilistic abstract interpretation, which allows us to measure the precision of an abstract domain and its operators by using vector spaces instead of partially order sets. However, the above mentioned works, do not define a general class of programs satisfying a bounded imprecision distance from all the possible concrete executions. It is worth remarking that, instead of defining domain-specific measures of the imprecision injected by program analysis, we define a general formal framework based on quasi-metric spaces and a weaker notion of completeness, called partial completeness, that allow us to control the amount of imprecision that we tolerate in the analysis. In fact, our work can be considered as a first step towards the extension of the results in [8, 44] to the case of partial completeness.

The approximation of grammar structures by abstract interpretation is not new. In [25] and [28] the authors introduced the idea of abstracting formal languages by abstract interpretation for the design of static analysers that manipulate symbolic structures. This provided both the source for new symbolic abstract domains for program analysis and the possibility of formalising known algorithms, such as parsers, as abstract interpreters. Abstractions into regular languages have been used in formal verification (e.g., see [17]). Moreover, in [40] Ganty et al. studied the language inclusion problem $L_1 \subseteq L_2$, where L_1 is regular, by relying on abstract interpretation. The authors approach checks whether an overapproximating abstraction of L_1, obtained by successively overapproximating the Kleene iterates of its least fixpoint characterization, is included in L_2. In program analysis non-regular approximations of formal languages have been used in aliasing analysis [36]. The idea of grammar abstraction as a relation between CF grammars has been also used for relating concrete and abstract syntax in [5]. None of the above mentioned approaches considered the more general problem of correlating languages in Chomsky's hierarchy by the theory of fixpoint abstraction by abstract interpretation.

Chapter 7

CONCLUSION

This thesis has investigated two parallel problems concerning the completeness property in abstract interpretation. In static program analysis, completeness is a desirable property but, unfortunately, it is very rare to acquire, while in abstracting indexed grammars, it is not helpful since it corresponds to the identity abstraction over the domain of indexed states. These two observations highlight two opposite interpretations about the completeness of abstractions in the abstract interpretation framework. We faced the two problems and we ended with the following results and new future perspectives.

Static Program Analysis

Because static analysis is incomplete by design, we weakened the notion of completeness into partial completeness. That is, we introduced a theoretical framework for defining the set of programs whose analysis on a given abstract domain has an ε-bounded level of imprecision, namely, the set of programs that are ε-partial complete for the considered abstract domain. This formal framework is based on a quasi-metric distance function on the elements of the abstract domain that respects the underlying structure of the domain. Among the other standard properties of quasi-metrics, we required a weak form of triangle inequality and the decidability of asking whether two abstract objects satisfy the quasi-metric over a fixed constant. We defined an abstract quasi-metric space as an abstract domain equipped with a compatible quasi-metric. We introduced the notion of ε-partial completeness class with respect to an abstract quasi-metric space as the set of all programs for which the abstract interpreter never outputs an abstract state whose (quasi-metric) distance from the concrete one exceeds ε. Interestingly, this class obeys similar properties as for the completeness case, indeed, it is an infinite non-extensional set for all ε-bound. We proved that, under the as-

sumption of unbounded imprecision, we can always find a program that exceeds a given ε-partial completeness class. This is a property of the quasi-metric that we use on our abstract domains of stores. Indeed, quasi-metrics A-compatible that always output the same constant value for every comparable elements, do not help us in quantifying the precision of an analysis. Then, we showed that the ε-partial completeness and incompleteness classes of a given non-ε trivial abstract domain are both non recursively enumerable. Moreover, the class of ε-partial incompleteness is productive. Therefore, it is possible to build a program that is able to enumerate all the programs in the ε-partial completeness (resp. incompleteness) class only if it coincides with the whole set of programs (resp. the empty set). Finally, we introduced a weaker class of partial completeness that admits limited imprecision only on a set of input traces. We proved that this new defined local ε-partial completeness class is r.e. if the abstract quasi-metric space of stores is ACC, therefore, there exists a (semi-decidable) program that can decide if a program is local ε-partial complete.

Partial completeness opens a new perspective in the field of static program analysis by abstract interpretation. As future work, we plan to (i) define a sound proof system that is able to prove whether a program satisfies the ε-partial completeness on a given abstract domain, and (ii) define the notion of partial completeness cliques, following the first definition in complexity theory [4], hence extending the results in [8] to the case of partial completeness cliques. As in the case of complete abstractions for a given program, also in the case of partial complete abstract domains the underlying lattice structure plays a central role. For this reason, we plan to study the existence of minimal domain transformers that can ensure the ε-partial completeness of the analysis with regard to a given program and constant ε, as done in [43] for standard completeness. Our work has a strong connection to code obfuscation. Code obfuscations are program transformations explicitly designed to degrade the results of program analysis, namely to induce imprecision, and therefore incompleteness in the analysis [42, 45, 46]. Being able to control or formalize the amount of imprecision induced by a code obfuscation technique in the results of a given analysis, could allow us to measure the efficiency of the obfuscation technique in confusing such analysis. This would be a very important way to quantify the efficiency of these techniques, which is still one of the main open challenges in software protection [14, 72]. Although the decidability requirement of the predicate $\delta_A(a, b) \leq \varepsilon$ resembles the decidability of a Blum complexity measure [7], our definition of quasi-metrics A-compatible as a measure of imprecision, is not a Blum complexity measure. We plan to deepen our study in order to formalize a measure of imprecision for abstract interpretation that satisfies the Blum's axioms in order to define partial completeness complexity classes.

Formal Languages

We reformulated the Chomsky's hierarchy between indexed and CF languages by using standard interpretation methods: we provided a fixpoint semantics for indexed languages and we characterized classes of less expressive languages in terms of fixpoint abstractions of this semantics. In our case, the approximation of indexed languages shows how it is possible to systematically and constructively derive all fixpoint descriptions for CF languages as abstract interpretations. Moreover, we showed how we can exploit incompleteness of the stack elimination abstraction on the indexed grammars, in order to construct the separation class between indexed and CF languages, namely, the set of all indexed languages which are not generated by a CF grammar.

We plan to generalize the above fixpoint abstraction by abstract interpretation over all the Chomsky's hierarchy and to formalize the known separation results between classes of languages as instances of incompleteness of language abstractions. We believe that a systematic reconstruction of Chomsky's hierarchy by fixpoint abstract interpretation may provide both new insights into a fundamental field of computer science and new algorithms and methods for approximating structures described by grammars. Indeed, the current work originated from the desire of finding suitable abstract domains for expressing the invariant properties among obfuscated malware variants [12, 32, 33]. Moreover, we plan to instantiate the partial completeness framework into the revised Chomsky's hierarchy by abstract interpretation, by using distances among, e.g., regular languages [69] as a measure for the incompleteness of the abstractions between the class of CF and regular languages.

REFERENCES

1. Jared Adams, Eric Freden, and Marni Mishna. From indexed grammars to generating functions. *RAIRO-Theoretical Informatics and Applications*, 47(4):325–350, 2013.
2. Alfred V. Aho. Indexed grammars – an extension of context-free grammars. *Journal of the ACM*, 15(4):647–671, 1968.
3. Alfred V. Aho. Nested stack automata. *Journal of the ACM*, 16(3):383–406, July 1969.
4. Andrea Asperti. The intensional content of rice's theorem. *ACM SIGPLAN Notices*, 43(1):113–119, 2008.
5. R. A. Ballance, J. Butcher, and S. L. Graham. Grammatical abstraction and incremental syntax analysis in a language-based editor. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 185–198, New York, NY, USA, 1988. ACM.
6. Eberhard Bertsch. On the relationship between indexed grammars and logic programs. *The Journal of Logic Programming*, 18(1):81–98, 1994.
7. Manuel Blum. A machine-independent theory of the complexity of recursive functions. *Journal of the ACM (JACM)*, 14(2):322–336, 1967.
8. Roberto Bruni, Roberto Giacobazzi, Roberta Gori, Isabel Garcia-Contreras, and Dusko Pavlovic. Abstract extensionality: on the properties of incomplete abstract interpretations. *PACMPL*, 4(POPL):28:1–28:28, 2020.
9. Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. A logic for locally complete abstract interpretations. In *Proc. 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2021)*, pages 1–13, 2021. Distinguished paper.
10. Marco Campion, Mila Dalla Preda, and Roberto Giacobazzi. Partial (in)completeness in abstract interpretation. Accepted for publication in *PACMPL (POPL 2022)*.
11. Marco Campion, Mila Dalla Preda, and Roberto Giacobazzi. Abstract interpretation of indexed grammars. In *International Static Analysis Symposium*, pages 121–139. Springer, 2019.
12. Marco Campion, Mila Dalla Preda, and Roberto Giacobazzi. Learning metamorphic malware signatures from samples. *Journal of Computer Virology and Hacking Techniques*, pages 1–17, 2021.
13. Ignacio Casso, José F Morales, Pedro López-García, Roberto Giacobazzi, and Manuel V Hermenegildo. Computing abstract distances in logic programs. In *International Symposium on Logic-Based Program Synthesis and Transformation*, pages 57–72. Springer, 2019.
14. Mariano Ceccato, Paolo Tonella, Cataldo Basile, Paolo Falcarin, Marco Torchiano, Bart Coppens, and Bjorn De Sutter. Understanding the behaviour of hackers while performing attack tasks in a professional setting and in a public challenge. *Empir. Softw. Eng.*, 24(1):240–286, 2019.

15. Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):137–167, June 1959.
16. E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.
17. Edmund M. Clarke, Orna Grumberg, and Somesh Jha. Verifying parameterized networks. *ACM Trans. Program. Lang. Syst.*, 19(5):726–750, 1997.
18. C. Collberg and J. Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional, 2009.
19. P. Cousot. Types as abstract interpretations (invited paper). In *Conference Record of the 24th ACM Symposium on Principles of Programming Languages* (*POPL '97*), pages 316–331. ACM Press, 1997.
20. P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theor. Comput. Sci.*, 277(1-2):47–103, 2002.
21. P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the 2nd International Symposium on Programming*, pages 106–130. Dunod, Paris, 1976.
22. P. Cousot and R. Cousot. Abstract interpretation frameworks. *J. Logic and Comput.*, 2(4):511–547, 1992.
23. P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation (Invited Paper). In M. Bruynooghe and M. Wirsing, editors, *Proc. of the 4th Internat. Symp. on Programming Language Implementation and Logic Programming* (*PLILP '92*), volume 631 of *Lecture Notes in Computer Science*, pages 269–295. Springer-Verlag, 1992.
24. P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *Conference Record of the 19th ACM Symp. on Principles of Programming Languages* (*POPL '92*), pages 83–94. ACM Press, 1992.
25. P. Cousot and R. Cousot. Compositional and inductive semantic definitions in fixpoint, equational, constraint, closure-condition, rule-based and game-theoretic form (Invited Paper). In P. Wolper, editor, *Proc. of the 7th Internat. Conf. on Computer Aided Verification* (*CAV '95*), volume 939 of *Lecture Notes in Computer Science*, pages 293–308. Springer-Verlag, 1995.
26. Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977.
27. Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 269–282, 1979.
28. Patrick Cousot and Radhia Cousot. Grammar semantics, analysis and parsing by abstract interpretation. *Theor. Comput. Sci.*, 412(44):6135–6192, 2011.
29. Patrick Cousot and Radhia Cousot. Abstract interpretation: past, present and future. In Thomas A. Henzinger and Dale Miller, editors, *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, pages 2:1–2:10. ACM, 2014.
30. Patrick Cousot, Roberto Giacobazzi, and Francesco Ranzato. Program analysis is harder than verification: A computability perspective. In *International Conference on Computer Aided Verification*, pages 75–95. Springer, 2018.
31. Federico Crazzolara. Quasi-metric spaces as domains for abstract interpretation. In Moreno Falaschi, Marisa Navarro, and Alberto Policriti, editors, *1997 Joint Conf. on Declarative Programming, APPIA-GULP-PRODE'97, Grado, Italy, June 16-19, 1997*, pages 45–56, 1997.

32. M. Dalla Preda, R. Giacobazzi, S. K. Debray, K. Coogan, and G. M. Townsend. Modelling metamorphism by abstract interpretation. In *Proc. of the 19th Int. Static Analysis Symp. (SAS '10)*, volume 6337 of *Lecture Notes in Computer Science*, pages 218–235. Springer-Verlag, Berlin, 2010.
33. Mila Dalla Preda, Roberto Giacobazzi, and Saumya K. Debray. Unveiling metamorphism by abstract interpretation of code properties. *Theor. Comput. Sci.*, 577:74–97, 2015.
34. D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Trans. Program. Lang. Syst.*, 19(2):253–291, 1997.
35. J. C. E. Dekker. Productive sets. *Trans. of the American Mathematical Society*, 78:129–149, 1955.
36. Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. *SIGPLAN Not.*, 29(6):230–241, June 1994.
37. Alessandra Di Pierro and Herbert Wiklicky. Measuring the precision of abstract interpretations. In *International Workshop on Logic-Based Program Synthesis and Transformation*, pages 147–164. Springer, 2000.
38. Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. Scaling static analyses at Facebook. *Commun. ACM*, 62(8):62–70, 2019.
39. A. Dovier and R. Giacobazzi. *Fondamenti dell'informatica. Linguaggi formali, calcolabilità e complessità*. Programma di mat. fisica elettronica. Bollati Boringhieri, 2020.
40. Pierre Ganty, Francesco Ranzato, and Pedro Valero. Language inclusion algorithms as complete abstract interpretations. In *International Static Analysis Symposium*, pages 140–161. Springer, 2019.
41. Gerald Gazdar. Applicability of indexed grammars to natural languages. In *Natural language parsing and linguistic theories*, pages 69–94. Springer, 1988.
42. R. Giacobazzi. Hiding information in completeness holes - new perspectives in code obfuscation and watermarking. In *Proc. of The 6th IEEE International Conferences on Software Engineering and Formal Methods (SEFM'08)*, pages 7–20. IEEE Press., 2008.
43. R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretation complete. *Journal of the ACM*, 47(2):361–416, March 2000.
44. Roberto Giacobazzi, Francesco Logozzo, and Francesco Ranzato. Analyzing program analyses. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 261–273. ACM, 2015.
45. Roberto Giacobazzi and Isabella Mastroeni. Making abstract interpretation incomplete: Modeling the potency of obfuscation. In *International Static Analysis Symposium*, pages 129–145. Springer, 2012.
46. Roberto Giacobazzi, Isabella Mastroeni, and Mila Dalla Preda. Maximal incompleteness as obfuscation potency. *Formal Aspects of Computing*, 29(1):3–31, 2017.
47. Seymour Ginsburg. *The Mathematical Theory of Context Free Languages.* McGraw-Hill Book Company, 1966.
48. John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. Introduction to automata theory, languages, and computation. *Acm Sigact News*, 32(1):60–65, 2001.
49. Harry B Hunt III. On the decidability of grammar problems. *Journal of the ACM (JACM)*, 29(2):429–447, 1982.
50. Sorin Istrail. Generalization of the Ginsburg-Rice Schützenberger fixed-point theorem for context-sensitive and recursive-enumerable languages. *Theoretical Computer Science*, 18(3):333–341, 1982.
51. Stephen Cole Kleene, NG De Bruijn, J de Groot, and Adriaan Cornelis Zaanen. *Introduction to metamathematics*, volume 483. van Nostrand New York, 1952.
52. Teodor Knapik, Damian Niwiński, and Paweł Urzyczyn. Higher-order pushdown trees are easy. In *International Conference on Foundations of Software Science and Computation Structures*, pages 205–222. Springer, 2002.

53. Naoki Kobayashi. Types and recursion schemes for higher-order program verification. In *Asian Symposium on Programming Languages and Systems*, pages 2–3. Springer, 2009.
54. Jeffrey C Lagarias. *The ultimate challenge: The 3x+ 1 problem*. American Mathematical Soc., 2010.
55. V. Laviron and F. Logozzo. Refining abstract interpretation-based static analyses with hints. In *Proc. of APLAS'09*, volume 5904 of *Lecture Notes in Computer Science*, pages 343–358. Springer-Verlag, 2009.
56. Francesco Logozzo, Corneliu Popeea, and Vincent Laviron. Towards a quantitative estimation of abstract interpretations. In *Workshop on Quantitative Analysis of Software*. Microsoft, 2009.
57. C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods Syst. Des.*, 6:11–44, 1995.
58. AN Maslov. Multilevel stack automata. *Problemy peredachi informatsii*, 12(1):55–62, 1976.
59. Antoine Miné. Tutorial on static inference of numeric invariants by abstract interpretation. *Foundations and Trends in Programming Languages*, 4(3-4):120–372, 2017.
60. Bruno Monsuez. System f and abstract interpretation. In *International Static Analysis Symposium*, pages 279–295. Springer, 1995.
61. J. Myhill. Creative sets. *Zeitschrift für Mathematische Logik und Grundilagen der Mathematik*, 1:97–108, 1955.
62. P. Odifreddi. *Classical Recursion Theory*. Studies in logic and the foundations of mathematics. Elsevier, 1999.
63. C-HL Ong. On model-checking trees generated by higher-order recursion schemes. In *21st Annual IEEE Symposium on Logic in Computer Science (LICS'06)*, pages 81–90. IEEE, 2006.
64. Barbara BH Partee, Alice G ter Meulen, and Robert Wall. *Mathematical methods in linguistics*, volume 30. Springer Science & Business Media, 2012.
65. E.L. Post. Recursively enumerable sets of positive integers and their decision problems. *Bulletin of the American Mathematical Society*, 50:284–316, 1944.
66. Mila Dalla Preda, Roberto Giacobazzi, and Niccolò Marastoni. Formal framework for reasoning about the precision of dynamic analysis. In David Pichardie and Mihaela Sighireanu, editors, *Static Analysis - 27th International Symposium, SAS 2020, Virtual Event, November 18-20, 2020, Proceedings*, volume 12389 of *Lecture Notes in Computer Science*, pages 178–199. Springer, 2020.
67. H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.*, 74(2):358–366, 1953.
68. H. Rogers. *Theory of recursive functions and effective computability*. The MIT press, 1992.
69. Ryoma Sin'ya. Asymptotic approximation by regular languages. In *International Conference on Current Trends in Theory and Practice of Informatics*, pages 74–88. Springer, 2021.
70. R. I. Soare. *Recursively Enumerable Sets and Degrees*. Springer-Verlag, 1980.
71. Pascal Sotin. Quantifying the precision of numerical abstract domains. 2010.
72. Bjorn De Sutter, Christian S. Collberg, Mila Dalla Preda, and Brecht Wyseur. Software protection decision support and evaluation methodologies (dagstuhl seminar 19331). *Dagstuhl Reports*, 9(8):1–25, 2019.
73. Arnaud Venet. Abstract cofibered domains: Application to the alias analysis of untyped programs. In *International Static Analysis Symposium*, pages 366–382. Springer, 1996.
74. Wallace Alvin Wilson. On quasi-metric spaces. *American Journal of Mathematics*, 53(3):675–684, 1931.
75. G. Winskel. *The formal semantics of programming languages: an introduction*. MIT press, 1993.